The Marriage of Univalence and Parametricity

NICOLAS TABAREAU, Gallinette Project-Team, Inria, France
ÉRIC TANTER, Computer Science Department (DCC), University of Chile, Chile
MATTHIEU SOZEAU, Gallinette Project-Team, Inria, France

Reasoning modulo equivalences is natural for everyone, including mathematicians. Unfortunately, in proof assistants based on type theory, which are frequently used to mechanize mathematical results and carry out program verification efforts, equality is appallingly syntactic and, as a result, exploiting equivalences is cumbersome at best. Parametricity and univalence are two major concepts that have been explored in the literature to transport programs and proofs across type equivalences, but they fall short of achieving seamless, automatic transport. This work first clarifies the limitations of these two concepts when considered in isolation, and then devises a fruitful marriage between both. The resulting concept, called *univalent parametricity*, is an extension of parametricity strengthened with univalence that fully realizes programming and proving modulo equivalences. Our approach handles both type and term dependency, as well as type-level computation. In addition to the theory of univalent parametricity, we present a lightweight framework implemented in the Coq proof assistant that allows the user to transparently transfer definitions and theorems for a type to an equivalent one, as if they were equal. For instance, this makes it possible to conveniently switch between an easy-to-reason-about representation and a computationally-efficient representation, as soon as they are proven equivalent. The combination of parametricity and univalence supports *transport à la carte*: basic univalent transport, which stems from a type equivalence, can be complemented with additional proofs of equivalences between functions over these types, in order to be able to transport more programs and proofs, as well as to yield more efficient terms. We illustrate the use of univalent parametricity on several examples, including a recent integration of native integers in Coq. This work paves the way to easier-to-use proof assistants by supporting seamless programming and proving modulo equivalences.

CCS Concepts: • **Theory of computation** → **Type theory**; **Type structures**; **Program reasoning**.

Additional Key Words and Phrases: Type Equivalence, Univalence, Parametricity, Proof Assistants, Coq

ACM Reference Format:
Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2020. The Marriage of Univalence and Parametricity. *J. ACM* 1, 1, Article 1 (January 2020), 45 pages.

1 INTRODUCTION

If mathematics is the art of giving the same name to different things, programming is the art of computing the same thing with different means. That sameness notion ought to be equivalence. Unfortunately, in programming languages as well as proof assistants such as Coq [Coq Development Team 2019] and Agda [Norell 2009], the notion of sameness or equality is appallingly syntactic. In dependently-typed languages that also serve as proof assistants, equivalences can be stated

*This work is partially funded by CONICYT FONDECYT Regular Project 1190058, CONICYT REDES Project 170067, ERC Starting Grant CoqHoTT 637339 and Inria Équipe Associée GECO.

Authors' addresses: Nicolas Tabareau, Gallinette Project-Team, Inria, Nantes, France; Éric Tanter, Computer Science Department (DCC), University of Chile, Santiago, Chile; Matthieu Sozeau, Gallinette Project-Team, Inria, Nantes, France.

0004-5411/2020/1-ART1
https://doi.org/

```
Inductive ℕ : Set :=
| O : ℕ
| S : ℕ → ℕ

Inductive Bin : Set :=
| O_Bin : Bin
| pos_Bin : positive → Bin

Inductive positive : Set :=
| xI : positive → positive
| x0 : positive → positive
| xH : positive
```

Fig. 1. Definition of $\mathbb{N}$ and Bin in Coq

and manually exploited, but they cannot be used as transparently and conveniently as syntactic or propositional equality. The benefits we ought to get from having equivalence as the primary notion of sameness include the possibility to state and prove results about a data structure (or mathematical object) that is convenient to formally reason about, and then automatically transport these results to other structures, for instance ones that are computationally more efficient, albeit less convenient to reason about.

Let us consider two equivalent representations of natural numbers available in the Coq proof assistant (Figure 1): Peano natural numbers $\mathbb{N}$, with constructors O and S, and binary natural numbers Bin, which denote a sequence of bits with a leading 1. Defining functions over $\mathbb{N}$ and reasoning about them is simple. For instance, $+_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ is a simple induction on the first argument, and proving that addition is commutative is similarly direct. Conversely, addition on binary natural numbers $+_{\text{Bin}} : \text{Bin} \to \text{Bin} \to \text{Bin}$ is defined with three functions—two mutually-recursive functions on positive and a simple function—making most reasoning much more involved. The other side of the comparison is that computing with $\mathbb{N}$ is much less efficient (if at all possible!) than computing with Bin. Ideally, one would want to apply easy inductive reasoning on $\mathbb{N}$ to establish properties of efficient functions defined on Bin.

The Challenge of Automatic Transport. An equivalence between $\mathbb{N}$ and Bin consists of two transport functions $\uparrow_{\mathbb{N}} : \text{Bin} \to \mathbb{N}$ and $\uparrow_{\text{Bin}} : \mathbb{N} \to \text{Bin}$ together with the proof that they are inverse of each other. Manually exploiting such an equivalence in order to transport properties on $\mathbb{N}$ to properties on Bin is however challenging, even for simple properties.

Consider the commutativity of addition on $\mathbb{N}$:

```
Definition +ℕ_comm : ∀ (n m : ℕ), n +ℕ m = m +ℕ n. (* simple inductive proof *)
```

from which one would like to deduce the commutativity of addition on Bin:

```
Definition +Bin_comm : ∀ (n m : Bin). n +Bin m = m +Bin n.
```

A proof of $+_{\text{Bin}}$_comm that exploits $+_{\mathbb{N}}$_comm and the $\mathbb{N}$-Bin equivalence would proceed as follows:

```
1. (↑ℕ n) +ℕ (↑ℕ m) = (↑ℕ m) +ℕ (↑ℕ n)  (* by +ℕ_comm (↑ℕ n) (↑ℕ m) *)
2. ↑Bin ((↑ℕ n) +ℕ (↑ℕ m)) = ↑Bin ((↑ℕ m) +ℕ (↑ℕ n)) (* by congruence *)
3. (↑Bin ↑ℕ n) +Bin (↑Bin ↑ℕ m) = (↑Bin ↑ℕ m) +Bin (↑Bin ↑ℕ n) (* ↑Bin is a monoid homomorphism *)
4. n +Bin m = m +Bin n (* by equivalence *)
```

Observe how one is forced to explicitly rewrite and reason about transports at each step. In particular, step 3 requires to show that $\uparrow_{\text{Bin}}$ is a monoid homomorphism between $+_{\mathbb{N}}$ and $+_{\text{Bin}}$. This

does not follow from the type equivalence between $\mathbb{N}$ and Bin, and therefore needs to be manually proven.

From this simple example, it is easy to imagine the difficulty of transporting entire libraries of structures and lemmas about their properties. The promise of *automatic transport across type equivalences* is to seamlessly allow users to operate with the most-suited representation as needed. In particular, deriving $+_{\mathsf{Bin}}$_comm ought to be as simple as transporting $+_{\mathbb{N}}$_comm (using a general transport operator ↑whose source and target types are inferred from context):

```
Definition +Bin_comm : ∀ (n m : Bin). n +Bin m = m +Bin n := ↑+ℕ_comm.
```

In the literature, two major concepts have been explored to achieve automatic transport across equivalences: *parametricity* and *univalence*. This article demonstrates that both of them are insufficient taken in isolation, and that it is possible to devise a marriage of univalence and parametricity that leverages both in order to fully realize programming and proving modulo equivalences.

Parametricity. Since the seminal work of Magaud and Bertot [2000] on translating proofs between different representations of natural numbers in Coq, there has been a lot of work in this direction, motivated by both program verification and mechanized mathematics, with several libraries available for either Isabelle/HOL [Huffman and Kunčar 2013] or Coq [Cohen et al. 2013; Zimmermann and Herbelin 2015]. At their core, these approaches build on the notion of parametricity [Reynolds 1983] and its potential for free theorems about observational equivalences [Wadler 1989], in order to obtain results such as data refinements for free [Cohen et al. 2013] and proofs for free [Bernardy et al. 2012].

Such a parametric transport is essentially a *white-box* approach that structurally rewrites observationally-equivalent terms. The previous example of Bin_comm can be handled by parametric transport. However, as we will demonstrate, the approach does not fully apply in the dependently-typed setting where computation at the type level is essential. (The Bin_comm example luckily does not rely on any type-level computation.)

Univalence. Univalence [Voevodsky 2010] is a novel principle for mathematics and type theory that postulates that equivalence is equivalent to equality. Leaving aside the most profound mathematical implications of Homotopy Type Theory (HoTT) and univalence [Univalent Foundations Program 2013], this principle should fulfill the promise of automatic transport of programs, theorems, and proofs across equivalences. There are currently two major approaches to realize univalence in a type theory. In Martin-Löf Type Theory (MLTT) [Martin-Löf 1975], and related theories such as the Calculus of (Inductive) Constructions [Coquand and Huet 1988; Paulin-Mohring 2015], univalence can only be expressed as an *axiom*. However, by the Curry-Howard correspondence, axioms have no computational content, since they correspond to free variables. Therefore an axiomatic general univalent transport is not effective. In concrete terms, this means that using axiomatic univalent transport will yield a "stuck term", stuck at the use of the axiom. There are several recent developments to build a dependent type theory with a computational account of univalence, most notably cubical type theories [Altenkirch and Kaposi 2015; Angiuli et al. 2018; Cohen et al. 2015], and concrete implementations such as Cubical Agda [Vezzosi et al. 2019] have started to appear. Cubical type theories fully achieve the consequences of realizing univalence, such as giving computational content to both functional and propositional extensionality.

Irrespective of how univalence is realized, *univalent transport* allows exploiting an equivalence between two types A and B in order to establish an equivalence between $P\ A$ and $P\ B$, for any arbitrary predicate P. But univalent transport alone does not address a major challenge for automatic

transport, namely that of inferring, from basic equivalences, the common predicate P out of arbitrarily complex dependent types. Additionally, while it is universally applicable as a *black-box* approach, univalent transport can yield unsatisfactory transported terms, as explained next.

Transport à la Carte. With univalent transport, one can always convert any development that uses $\mathbb{N}$ into one that uses Bin, both in computationally-relevant parts and in parts that deal with reasoning and formal properties. However, univalent transport does not necessarily reconcile ease of reasoning with efficient computation. Indeed, univalently transporting a function $\mathbb{N} \to \mathbb{N}$ to a function $\mathsf{Bin} \to \mathsf{Bin}$ yields a function that first converts its binary argument to a natural number, performs the original (slow) computation and finally converts the result back to a binary number. Dually, if one starts from a $\mathsf{Bin} \to \mathsf{Bin}$ function and transports it to $\mathbb{N} \to \mathbb{N}$, the resulting function will still execute efficiently, but simple $\mathbb{N}$-based inductive reasoning will not be applicable to it.

The problem is that univalent transport across the $\mathbb{N}$-Bin equivalence does not magically exploit the correspondence between different *implementations* of functions that operate on these types, such as between $+_{\mathbb{N}}$ and $+_{\mathsf{Bin}}$. Such term-level correspondences *are* exploited in parametricity-based approaches, and require additional proof and engineering effort.

Therefore, in addition to addressing the limitations of parametricity and univalence when taken in isolation, an essential component of automatic transport is the tradeoff between the cost of manually establishing equivalences (between both types and functions that operate on them), and the ease of automatic univalent transport. One wishes for an automatic transport mechanism *à la carte*, which exploits user-provided equivalences between terms when available, and falls back to univalent transport otherwise.

Univalent Parametricity. This article proposes a marriage of univalence and parametricity that enables automatic transport à la carte across equivalences. It deeply connects and intertwines (white-box) parametric transport and (black-box) univalent transport in a fruitful manner. Essentially, univalent parametricity is a strengthening of the parametricity translation for dependent types that demands the relation on the universe to be compatible with equivalences. This paper is structured as follows. We first recall parametricity in type theory and present its limitations to transport definitions across equivalences (§2). We then proceed similarly with univalence, highlighting the complementarity between both approaches (§3). Next, we illustrate how univalent parametricity achieves seamless automatic transport across equivalences from a user point of view (§4). We develop the theory of univalent parametricity for the Calculus of Constructions with universes CC_ω (§5), and for the Calculus of Inductive Constructions CIC (§6). We present the realization of univalent parametricity in the Coq proof assistant as a shallow embedding that exploits the typeclass mechanism (§7). Note that this implementation in Coq does not give any computational content to univalence and extensionality axioms; instead, it brings automatic univalent transport to programmers, using such axioms sparingly in order to disrupt computation as little as possible. We explain how the illustration of §4 is effectively implemented in Coq (§8). We end by describing a case study related to the recent integration of native integers in Coq (§9). §10 discusses related work, and §11 concludes.

The complete Coq development (compatible with Coq v8.10) is available online:

https://github.com/coqhott/univalent_parametricity

Prior publication. This paper is a substantial extension of a prior conference publication [Tabareau et al. 2018]. First, we explain in details the limitations of both parametricity and univalence when considered in isolation (§2-§3). Second, we extend our original proposal to integrate user-defined correspondences between terms of different (related) types, which is absolutely necessary to reconcile ease of reasoning and efficient computation, and to realize transport à la carte. The

$$
\begin{aligned}
[\![\mathsf{Type}_i]\!]_p\ A\ B &\triangleq A \to B \to \mathsf{Type}_i \\
[\![\Pi a : A.B]\!]_p\ f\ g &\triangleq \Pi(a : A)(a' : A')(a^\varepsilon : [\![A]\!]_p\ a\ a').[\![B]\!]_p\ (f\ a)\ (g\ a') \\
[\![x]\!]_p &\triangleq x^\varepsilon \\
[\![\lambda x : A.t]\!]_p &\triangleq \lambda(x : A)(x' : A')(x^\varepsilon : [\![A]\!]_p\ x\ x').[\![t]\!]_p \\
[\![t\ u]\!]_p &\triangleq [\![t]\!]_p\ u\ u'\ [\![u]\!]_p \\
\\
[\![\cdot]\!]_p &\triangleq \cdot \\
[\![\Gamma, x : A]\!]_p &\triangleq [\![\Gamma]\!]_p, x : A, x' : A', x^\varepsilon : [\![A]\!]_p\ x\ x'
\end{aligned}
$$

Fig. 2. Parametricity translation for CC_ω (from [Bernardy et al. 2012])

illustrations of §4 and §8 are therefore novel as well, as they take advantage of this new feature, in addition to providing a detailed user perspective on the Coq framework. §7 is extended accordingly to deal with transport à la carte. §5 clarifies the two reasoning principles, white-box and black-box, supported by univalent parametricity, and explains how they support transport à la carte. Finally, the case study of reasoning about/with native integers (§9) is entirely new.

2 PARAMETRICITY IS NOT ENOUGH

We first review the development of parametricity in dependently-typed theories (§2.1), and discuss its use and limitations for transporting some programs and proofs (§2.2). Finally, we consider an extension of parametricity that addresses some limitation, but is still limited when type-level computation is involved (§2.3).

2.1 Parametricity for Dependent Types

Reynolds originally formulated the relational interpretation of types to establish parametricity of System F [Reynolds 1983]. More recently, Bernardy et al. [2012] generalized the approach to pure type systems, including the Calculus of Constructions with universes CC_ω, and its extension with inductive types, the Calculus of Inductive Constructions CIC, which is at the core of proof assistants like Coq.[1]

The syntax of CC_ω includes a hierarchy of universes Type_i, variables, applications, lambda expressions and dependent function types:

$$A, B, M, N ::= \mathsf{Type}_i \mid x \mid M\ N \mid \lambda x : A.\ M \mid \Pi x : A.\ B$$

Its typing rules are standard, and hence omitted here—see Paulin-Mohring [2015] for a recent presentation.

Parametricity for CC_ω can be defined as a logical relation $[\![A]\!]_p$ for every type A. Specifically, $[\![A]\!]_p\ a_1\ a_2$ states that a_1 and a_2 are related at type A. The essence of Bernardy et al.'s approach is to express parametricity as a translation from terms to the expression of their relatedness *within* the same theory; indeed, the expressiveness of CC_ω allows the logical relation to be stated in CC_ω itself. Note that because terms and types live in the same world, $[\![-]\!]_p$ is defined for every term.

[1] CC_ω features a predicative hierarchy of universes Type_i, and also an impredicative universe Prop. In this paper, we focus on the predicative hierarchy, because adding an impredicative universe has little impact. Section 5.2.1 explains the minor changes for integrating the impredicative universe Prop.

Figure 2 presents the definition of $[\![-]\!]_p$ for CC_ω, based on the work of Bernardy et al. [2012]. For the universe Type_i, the translation is naturally defined as (arbitrary) binary relations on types. For the dependent function type $\Pi a : A.B$, the translation specifies that related inputs at A, as witnessed by e, yield related outputs at B.

Following Bernardy et al., the prime notation (*e.g.,* A') denotes duplication with renaming, where each free variable x is replaced with x'. Similarly, the translation of a lambda term $\lambda x : A.t$ is a function that takes two arguments and a witness x^ε that they are related; a variable x is translated to x^ε; a translated application passes the original argument, its renamed duplicate, along with its translation, which denotes the witness of its self-relatedness. The translation of type environments follows the same augmentation pattern, with duplication-renaming of each variable as well as the addition of the relational witness x^ε.

Armed with this translation, it is possible to prove an abstraction theorem à la Reynolds, saying that a well-typed term is related to itself (more precisely, to its duplicated-renamed self):

Theorem 2.1 (Abstraction theorem). *If* $\Gamma \vdash t : A$ *then* $[\![\Gamma]\!]_p \vdash [\![t]\!]_p : [\![A]\!]_p\ t\ t'$.

In particular, this means that the translation of a term $[\![t]\!]_p$ is itself the *proof* that t is relationally parametric.

The abstraction theorem is proven by showing the fundamental property of the logical relation for each constructor of the theory. In particular, for the cumulative hierarchy of universes, $\vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}$. This means that we have a kind of fixpoint property for the relation on Type_i:

$$\vdash [\![\mathsf{Type}_i]\!]_p : [\![\mathsf{Type}_{i+1}]\!]_p\ \mathsf{Type}_i\ \mathsf{Type}_i.$$

For parametricity, this property holds because the following is a proof term:

$$\lambda(A\ B : \mathsf{Type}_i).\ \mathsf{Type}_i : \mathsf{Type}_i \to \mathsf{Type}_i \to \mathsf{Type}_{i+1}.$$

Note that this necessary fixpoint property is not necessarily trivial to satisfy in any variant of parametricity, as we will see later (§5).

2.2 Using Parametricity to Transport Programs and Proofs

The parametricity translation together with the abstraction theorem (Theorem 2.1) are powerful tools to derive free theorems (and proofs) [Bernardy et al. 2012]. However, the abstraction theorem is only concerned with what we can call *reflexive homogeneous* instances of the logical relation, *i.e.,* relating a term with itself (*i.e.,* reflexive) and hence at the same type (*i.e.,* homogeneous). Thus, in order to be able to relate functions and theorems over different types, such as $\mathbb{N}$ and Bin, the standard solution is to define functions manipulating a common abstraction of their algebraic structure—in the case of natural numbers, a type with a zero and a successor function—together with an elimination principle. Then, by parametricity, we know that such a function defined on the common abstraction behaves the same if we instantiate it with $\mathbb{N}$ or Bin, because it must preserve any relation between $\mathbb{N}$ and Bin, in particular equivalences. This is for instance the approach taken in the CoqEAL framework [Cohen et al. 2013].

Parametricity presents two important issues, which we call the *anticipation problem* and the *computation problem*. The anticipation problem is that, in order to reap the benefits of parametricity to transport programs and proofs, one must anticipate and explicitly exhibit *a priori* an interface that is common to the types dealt with, and to define all functions generically in terms of this common interface. Engineering-wise, this anticipation might be problematic. Furthermore, defining the right interface might be challenging. Of course, in the case of the addition on natural numbers[2],

[2]This function definition corresponds to the infix notation $+_{\mathbb{N}}$ used in §1.

it is fairly straightforward to define the common interface, because the definition of `plus` is only using the successor function and the eliminator.

```
Definition plus (n m : ℕ) : ℕ := ℕ_rect (fun _ ⇒ ℕ) m (fun _ res ⇒ S res) n.
```

Similarly, as already noticed by Cohen et al. [2013], finding the right interface is direct when dealing with primitive inductive types, but it becomes quite challenging when dealing with types defined using a combination of several type constructors.

The computation problem is that parametricity does not scale to computation at the type level. To illustrate this, consider the proof that `0` is different from `S n`, for every natural number `n`.

```
Definition diff n (e : 0 = S n) : False :=
  let P_ℕ := ℕ_rect (fun _ ⇒ Type) (0 = 0) (fun n _ ⇒ False) in
  eq_rect ℕ 0 (fun n' _ ⇒ P_ℕ n') (eq_refl 0) (S n) e.
```

This definition uses the elimination principle of equality over a predicate that is defined by elimination on natural numbers. It typechecks because in the branch for `0`, $\mathsf{P}_{\mathbb{N}}\ 0$ reduces to `0 = 0` and in the branch `S n, e`, $\mathsf{P}_{\mathbb{N}}\ (S\ n)$ reduces to `False`.

Now, if one tries to generalize this definition of this function by making it modular with respect to any type of natural numbers with zero, successor and a constant for the elimination principle, the result is ill-typed, because the abstraction `P_abs` of $\mathsf{P}_{\mathbb{N}}$ does not compute and so `P_abs 0` is not *definitionally* equal to `0 = 0`. This issue can be sidestepped by adding the computational laws of the eliminator on natural numbers as *propositional* equalities in the generalized version. But then, one needs to deal with rewriting explicitly where otherwise everything was handled implicitly by conversion.[3] This rewriting phase is not at all handled by parametricity.

2.3 Heterogeneous Parametricity Translation

To address the anticipation problem described above, we would like to be able to relate $\mathbb{N}$ or `Bin` *directly—i.e.,* without relying on a common interface that captures their algebraic structure—simply because they are equivalent as types.

Observe that the definition of the parametricity translation of Bernardy et al. given in Figure 2 is *homogeneous*, in that terms are related *at the same type*, *i.e.,* $[\![A]\!]_p\ a_1\ a_2$. This allows us to provide instances of the parametricity relation such as $[\![\mathsf{Type}_i]\!]_p\ \mathbb{N}\ \mathsf{Bin}$. But once $\mathbb{N}$ and `Bin` are related as types, we will want to relate some of their inhabitants, such as `0` and 0_{Bin}, which means we also need to consider *heterogeneous* instances, *i.e.,* over terms of different (related) types.

But actually, parametricity itself is eminently heterogeneous: modifying the parametricity translation to reflect this is, in fact, straightforward. It suffices to additionally consider a global context Ξ of defined constant triples, where each triple consists of two constants (such as `0` and 0_{Bin}) and a witness that they are parametrically related. The global context Ξ is defined as the following

[3] Note that this issue appears because we are working with an intensional type theory. It would not be present in an extensional type theory, but in this work we only consider theories with a decidable type checking algorithm.

telescope Ξ_n:

$$\Xi_0 = \cdot$$
$$\Xi_1 = (c_1^\circ : A_1^\circ;\ c_1^\bullet : A_1^\bullet;\ c_1^\otimes : [\![A_1]\!]_p^{\Xi_0}\ c_1^\circ\ c_1^\bullet)$$
$$\ldots$$
$$\Xi_n = \Xi_{n-1}, (c_n^\circ : A_n^\circ;\ c_n^\bullet : A_n^\bullet;\ c_n^\otimes : [\![A_n]\!]_p^{\Xi_{n-1}}\ c_n^\circ\ c_n^\bullet)$$

Note that in the definition above, we have extended the parametricity translation to additionally take the global context into account, $[\![\cdot]\!]_p^\Xi$. The definition of Figure 2 is accordingly extended on constants as follows:

$$[\![c^\circ]\!]_p^\Xi \triangleq c^\otimes \text{ when } (c^\circ : \_\ ;\ c^\bullet : \_\ ;\ c^\otimes : \_) \in \Xi$$

We note $|\Xi|_\circ$, $|\Xi|_\bullet$, and $|\Xi|_\varepsilon$, the typing contexts obtained by projecting the respective components of each triple of Ξ, and $|\Xi|$ the whole typing context $|\Xi|_\circ, |\Xi|_\bullet, |\Xi|_\varepsilon$.

To state the fundamental theorem of parametricity in this setting, we need to define a notion of *white box* translation of a given term, $[a]_\square^\Xi$, which is essentially the identity function except for constants in Ξ, which are translated as

$$[c^\circ]_\square^\Xi \triangleq c^\bullet \text{ when } (c^\circ : \_\ ;\ c^\bullet : \_\ ;\ c^\otimes : \_) \in \Xi$$

We can now state the "White Box" Fundamental Property (FP): when a term a is well-typed in the context $|\Xi|_\circ$, its white box translation $[a]_\square^\Xi$ is also well-typed in the context $|\Xi|_\bullet$, and the parametricity translation of a provides a witness that a is related to $[a]_\square^\Xi$.

Corollary 2.2 (White Box Fundamental property). *If* $|\Xi|_\circ \vdash a : A$ *then* $|\Xi|_\bullet \vdash [a]_\square^\Xi : [A]_\square^\Xi$ *and* $|\Xi| \vdash [\![a]\!]_p^\Xi : [\![A]\!]_p^\Xi\ a\ [a]_\square^\Xi$.

Proof. From a (global) context $|\Xi|$, one can construct a substitution σ from $|\Xi|$ to $[\![|\Xi|_\circ]\!]_p$ by associating the triple $(c^\circ, c^\bullet, c^\otimes) \mapsto (c^\circ, c^{\circ\prime}, c^{\circ\varepsilon})$. We note $\sigma(t)$ the application of the substitution σ to a term t. By the abstraction theorem, we have that

$$[\![|\Xi|_\circ]\!]_p \ \vdash [\![a]\!]_p : [\![A]\!]_p\ a\ a'.$$

The corollary follows from the fact that $\sigma([\![a]\!]_p) \equiv [\![a]\!]_p^\Xi$ and $\sigma(a') \equiv [a]_\square^\Xi$. □

Note that this property is only valid in a closed world (*i.e.,* no variables but potentially global constants), because, as parametricity is not internalized in the theory, there is no witness that variables or (equivalently) axioms are parametric.

The introduction of the global context Ξ allows us to provide a direct heterogeneous extension to the parametricity translation, however mentioning it explicitly in the translation makes the notation heavy. To avoid mentioning the global context explicitly, we introduce the notation

$$a \approx_p b : A \bowtie_p B \triangleq [\![A]\!]_p^\Xi\ a\ b$$

which relates two terms a and b at two related—but potentially different—types A and B. Of course, this notation only makes sense when $B = [A]_\square^\Xi$ in the current global context Ξ, and we implicitly assume it is the case when we use this notation. The definition of the parametricity translation of Figure 2 for closed terms is the special homogeneous case, where there is no global context and where terms are related at the same type, *i.e.,* $[\![A]\!]_p a_1\ a_2$ corresponds to $a_1 \approx_p a_2 : A \bowtie_p A$.

The heterogeneous version of the parametricity translation makes it possible to relate terms of different types, such as $\mathsf{0} \approx_p \mathsf{0}_{\mathsf{Bin}} : \mathbb{N} \bowtie_p \mathsf{Bin}$, assuming that $\mathbb{N} \approx_p \mathsf{Bin} : \mathsf{Type} \bowtie_p \mathsf{Type}$ appears in the global context. Hereafter, whenever $a \approx_p b : A \bowtie_p B$ holds, we say that a and b are

parametrically related. With the notation that makes the global context implicit, the parametricity relation on dependent functions can be expressed as:

$$f \approx_p g : \Pi a : A.P\ a \bowtie_p \Pi b : B.Q\ b \triangleq \Pi\ (a : A)\ (b : B).\ a \approx_p b : A \bowtie_p B \to f\ a \approx_p g\ b : P\ a \bowtie_p Q\ b$$

assuming $A \approx_p B : \mathsf{Type} \bowtie_p \mathsf{Type}$ and $P \approx_p Q : A \to \mathsf{Type} \bowtie_p B \to \mathsf{Type}$.

For instance, to show that $\mathbb{N}$ and Bin are parametrically related, one needs to provide a relation $\mathsf{R}_{\mathbb{N}\mathsf{Bin}}$ between $\mathbb{N}$ and Bin. While there are several equivalent ways of defining this relation, the canonical one reuses the transport function $\uparrow_{\mathbb{N}}$ that comes from the equivalence between $\mathbb{N}$ and Bin:

```
Definition R_NBin := fun n m ⇒ n = ↑_N m.
```

Then, in order to make explicit that Bin behaves the same as $\mathbb{N}$, one can define a successor function $\mathsf{S}_{\mathsf{Bin}} : \mathsf{Bin} \to \mathsf{Bin} := \mathsf{fun}\ \mathsf{n} \Rightarrow \mathsf{n} + 1$ and show that 0 and 0_{Bin} are parametrically related in the global context which contains ($\mathbb{N}$,Bin,$\mathsf{R}_{\mathbb{N}\mathsf{Bin}}$), as well as S and $\mathsf{S}_{\mathsf{Bin}}$, which amounts to providing inhabitants for the following types:

$$0^{\circledast} : 0 \approx_p 0_{\mathsf{Bin}} : \mathbb{N} \bowtie_p \mathsf{Bin} \quad \triangleq \quad 0 = \uparrow_{\mathbb{N}} 0_{\mathsf{Bin}}$$

$$\mathsf{S}^{\circledast} : \mathsf{S} \approx_p \mathsf{S}_{\mathsf{Bin}} : \mathbb{N} \to \mathbb{N} \bowtie_p \mathsf{Bin} \to \mathsf{Bin} \quad \triangleq \quad \forall\ n\ m,\ n = \uparrow_{\mathbb{N}} m \to \mathsf{S}\ n = \uparrow_{\mathbb{N}} (\mathsf{S}_{\mathsf{Bin}}\ m)$$

Additionally, one also needs to show that Bin satisfies an induction principle corresponding to the induction principle of $\mathbb{N}$. Recall that the induction principle of $\mathbb{N}$ has type

```
Definition N_rect: ∀ P : N → Type, P 0 → (∀ n, P n → P (S n)) → ∀ n : N, P n.
```

Thus, the corresponding induction principle for Bin ought to have type

```
Definition N_rect_Bin : ∀ P : Bin → Type, P 0_Bin → (∀ n, P n → P (S_Bin n)) → ∀ n : Bin, P n.
```

Note that this induction principle is very different from `Bin_rect`, the canonical induction principle derived for the inductive definition of Bin (Figure 1). Finally, we also need to prove that $\mathbb{N}\texttt{\_rect}_{\mathsf{Bin}}$ is parametrically related to $\mathbb{N}$`_rect` in the global context:

$$\Xi_{\mathbb{N}} = (\mathbb{N};\mathsf{Bin};\mathsf{R}_{\mathbb{N}\mathsf{Bin}}), (0;0_{\mathsf{Bin}};0^{\circledast}), (\mathsf{S};\mathsf{S}_{\mathsf{Bin}};\mathsf{S}^{\circledast})$$

Using these definitions, it becomes possible to use parametricity to automatically convert a definition over $\mathbb{N}$ that uses the induction principle $\mathbb{N}$`_rect` to an equivalent one over Bin. For instance, consider the definition of `plus` defined on $\mathbb{N}$. By Corollary 2.2, it is possible to automatically deduce that the function

```
Definition plus_NBin (n m : Bin) : Bin := N_rect_Bin (fun _ ⇒ Bin) m (fun _ res ⇒ S_Bin res) n.
```

is parametrically related to `plus`, and thus behaves in the same way, because $\mathtt{plus}_{\mathbb{N}\mathsf{Bin}}$ is equal to $[\mathtt{plus}]^{\Xi_{\mathbb{N}}}_{\square}$. This means that for all n m : $\mathbb{N}$ and n' m' : Bin, the following holds:

```
n ≈_p n' : N ⋈_p Bin → m ≈_p m' : N ⋈_p Bin → plus n m ≈_p plus_NBin n' m' : N ⋈_p Bin.
```

That is, using the heterogeneous version of the parametricity translation addresses the anticipation problem identified above, because we do not need to rely on a common interface between $\mathbb{N}$

and Bin and define all functions over this interface, as would be required in CoqEAL [Cohen et al. 2013].

The Limits of Parametricity. However, using the heterogeneous parametricity translation to obtain automatic transport still does not scale to dependent types, because of the computational problem of parametricity: parametrically-related functions behave the same *propositionally* but not *definitionally*.

Let us go back to the `diff` example of Section 2.2. Using the white box FP to get a parametrically-related definition of `diff` over Bin, we could expect to get

```
Fail Definition diff_Bin n (e : O_Bin = S_Bin n) : False :=
 let P_Bin := ℕ_rect_Bin (fun _ ⇒ Type) (O_Bin = O_Bin) (fun n _ ⇒ False) in
 eq_rect Bin O_Bin (fun n' _ ⇒ P_Bin n') (eq_refl O_Bin) (S_Bin n) e.
```

But this term does not typecheck, as the Coq error message explains:

```
The term "..." has type "P_Bin (S_Bin n)" while it is expected to have type "False".
```

This is because even though ℕ_rect and ℕ_rect_Bin are parametrically related, they are not equal by conversion. And indeed, the equality ℕ_rect_Bin _ P₀ PS (S_Bin n) = PS n only holds propositionally, but not definitionally. This means that the premise of Corollary 2.2 does not hold here: the term `diff` does not typecheck in the context $\Xi_{\mathbb{N}}$ where each constant, and in particular the constant ℕ_rect, is considered as a black box and does not come with associated computational rules. So, while moving to a heterogeneous presentation seems promising with respect to the anticipation problem, it is insufficient to deal with the computation problem of parametricity.

3 UNIVALENCE IS NOT ENOUGH

We now briefly review the notion of type equivalences (§3.1) and the univalence principle (§3.2), and explain why univalence alone is not sufficient for automatic transport across equivalences (§3.3).

3.1 Type Equivalences

A function $f : A \to B$ is an *equivalence* iff there exists a function $g : B \to A$ together with proofs that f and g are inverse of each other. More precisely, the *section* property states that $\forall a : A, g(f(a)) = a$, and the *retraction* property dually states that $\forall b : B, f(g(b)) = b$. An additional condition between the section and the retraction, called here the *adjunction condition*, expresses that the equivalence is uniquely determined by the function f (and hence that being an equivalence is proof irrelevant).

Definition 3.1 (Type equivalence). Two types A and B are equivalent, noted $A \simeq B$, iff there exists a function $f : A \to B$ that is an equivalence.

A type equivalence therefore consists of two *transport functions* (*i.e.,* f and g), as well as three properties. The transport functions are obviously computationally relevant, because they actually construct terms of one type based on terms of the other type. Note that from a computational point of view, there might be different ways to witness the equivalence between two types, which would yield different transports.

Armed with a type equivalence $A \simeq B$, one can therefore *manually* port a library that uses A to a library that uses B, by using the $A \to B$ function in covariant positions and the $B \to A$ function in contravariant positions. However, with type dependencies, all uses of transport at the term level can leak at the type level. This leakage requires not only the use of sections or retractions to deal

with type mismatches, but also additional properties relating existing functions, as illustrated in §1 with the fact that the equivalence is a homomorphism with respect to the addition on natural numbers.

This also means that while the properties of an equivalence are not used computationally for transporting from A to B or vice versa, their computational content can matter when one wants to exploit the equivalence of constructors that are indexed by A or by B. For instance, to establish that a term of type $T\ (g(f(a))$ actually has type $T\ a$, one needs to rewrite the term using the section of the equivalence—which means applying it as a (computationally-relevant) function.

3.2 Univalence

Univalence is a principle that aligns type equivalence with propositional equality [Voevodsky 2010].

Definition 3.2 (Univalence). For any two types A, B, the canonical map $(A = B) \to (A \simeq B)$ is an equivalence.

In particular, this means that $(A = B) \simeq (A \simeq B)$. Therefore, univalence allows us to generalize Leibniz's principle of indiscernibility of identicals, to what we call the principle of *Indiscernibility of Equivalents*.

Theorem 3.3 (Indiscernibility of Equivalents). *For any $P : \mathsf{Type} \to \mathsf{Type}$, and any two types A and B such that $A \simeq B$, we have $P\ A \simeq P\ B$.*

Proof. Direct using univalence: $A \simeq B \implies A = B \implies P\ A = P\ B \implies P\ A \simeq P\ B$ □

In particular, univalence promises immediate transport: if A and B are equivalent, then we can always convert some $P\ A$ to some (equivalent) $P\ B$, for every inhabitant of $P\ A$, even axioms.

Corollary 3.4 (Black Box Fundamental Property). *For any $P : \mathsf{Type} \to \mathsf{Type}$, and any two types A and B such that $A \simeq B$, there exists a function $\uparrow_{\blacksquare} : P\ A \to P\ B$.*

We call this result the "Black Box" Fundamental Property because it can be used to blindly transport a term of type $P\ A$ to a term of type $P\ B$, without looking at its particular syntactical structure. As such, it is very useful to solve the computational issue of parametricity.

Realizing Univalence. In CIC and MLTT, univalence cannot be proven and is therefore defined as an *axiom*. Because the proof of Theorem 3.3 starts by using the univalence axiom to replace type equivalence with propositional equality, before proceeding trivially with rewriting, it has no computational content, and hence we cannot exploit (axiomatic) univalence to reap the benefits of automatic transport of programs and their properties across equivalent types. It is important for transport to be *effective*, *i.e.,* that it has computational content.

Intuitively, an effective function ensures *canonicity*: it never gets stuck due to the use of an axiom. Conversely, a function that uses an axiom and hence "does not compute" is called *ineffective*. By extension, a type equivalence $A \simeq B$ consisting of two functions $f : A \to B$ and $g : B \to A$ is said to be *effective* iff both f and g are effective functions.

To solve the issue of effectiveness, Cubical Type Theory has recently been proposed [Cohen et al. 2015; Vezzosi et al. 2019]. This theory is an extension of MLTT in which n-dimensional cubes can be directly manipulated, making it possible to define a notion of equality between two terms as the type of the line (1-dimensional cube) between those two terms. This way, the induced notion of equality is more extensional than the usual Martin-Löf identity type, and it satisfies univalence computationally, so the induced transports are effective.

```
addp  : PathP (λ i → ℕ≡Bin i → ℕ≡Bin i → ℕ≡Bin i) _+_ _+Bin_
addp i = transp (λ j → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j)) (~ i) _+_

+Bin-assoc : (m n o : Bin) → m +Bin (n +Bin o) ≡ (m +Bin n) +Bin o
+Bin-assoc
  = transport (λ i → (m n o : ℕ≡Bin i) → addp i m (addp i n o) ≡ addp i (addp i m n) o)
              +-assoc
```

Fig. 3. Transporting associativity from unary to binary naturals, in Cubical Agda (from [Vezzosi et al. 2019])

3.3 Univalence vs. Automatic Lifting

However, even when it is effective, univalence *alone* is not enough to support the automatic transport of functions that are defined on equivalent types.

Let us go back to the example of addition on natural numbers. There exists a complicated but efficient definition of addition on binary natural numbers, $\texttt{plus}_{\texttt{Bin}}$:[4]

```
Definition plus_Bin (n m : Bin) : Bin := (* complex definition *).
```

Showing most properties of $\texttt{plus}_{\texttt{Bin}}$, such as associativity and commutativity, is much more involved than their counterparts on $\mathbb{N}$. Ideally, after proving once and for all that $\texttt{plus}_{\texttt{Bin}}$ is "equal" to plus, one would like to be able to obtain these theorems for free by transporting the proofs for plus on $\mathbb{N}$, *i.e.,* rewriting through this "equality".

The problem is that even in a univalent type theory, $\texttt{plus}_{\texttt{Bin}}$ and plus cannot be proven equal directly, because they are not defined on the same type. Indeed, the "equality" between $\texttt{plus}_{\texttt{Bin}}$ and plus is *heterogeneous* and only makes sense because there is an equivalence between $\mathbb{N}$ and Bin. This means that technically, the actual equality $\texttt{e}_{\mathbb{N}\texttt{Bin}}$ that can be stated and proven is between the pairs ($\mathbb{N}$; plus) and (Bin; $\texttt{plus}_{\texttt{Bin}}$) at the telescope type Σ (A : Type), (A → A → A). Then to transport the proof of commutativity of plus

```
Definition plus_comm : ∀ (n m : ℕ), plus m n = plus n m.
```

to a proof of commutativity of $\texttt{plus}_{\texttt{Bin}}$, one needs to exhibit the predicate

```
P_comm := fun X _ ⇒ ∀ (n m : X.1), X.2 m n = X.2 n m
```

to be passed to the eliminator of equality in order to define $\texttt{plus}_{\texttt{Bin}}\texttt{\_comm}$ as

```
Definition plus_Bin_comm : ∀ (n m:Bin), plus_Bin m n = plus_Bin n m :=
  eq_rect (Σ (A : Type), (A → A → A)) (ℕ; plus) P_comm plus_comm (Bin; plus_Bin) e_ℕBin
```

This generalization step, which can quickly become complex, cannot in general be automatically inferred, and so needs to be explicitly provided by the user.

In Cubical Type Theory [Vezzosi et al. 2019], one would rather rely on the primitive notion of dependent path and transport the proof as depicted in Figure 3. In that case, one must still explicitly specify an abstraction of the lemma statement to give to the transport function, and produce a dependent path between the two notions of addition. In this example, the addition on binary

[4]This function definition corresponds to the infix notation $+_{\texttt{Bin}}$ used in §1.

numbers _+Bin_ is defined by a "naive" transport of the addition on unary numbers. The direct proof addp shows that it is related to addition of unary numbers through the type equivalence ℕ≡Bin. Note that the proof of this equality for the efficient addition on binary numbers would be much more involved, but this is not the point here. The point we want to stress is that transporting the proof term +-assoc to the proof term +Bin-assoc requires the user to exhibit the predicate $\lambda i \to (m\ n\ o : \mathbb{N} \equiv \mathsf{Bin}\ i) \to \mathsf{addp}\ i\ m\ (\mathsf{addp}\ i\ n\ o) \equiv \mathsf{addp}\ i\ (\mathsf{addp}\ i\ m\ n)\ o$. We believe that, even in this simple example, a user would arguably appreciate some help from an automatic tool.

This issue is very similar to the anticipation problem of parametricity described in Section 2.2. Indeed, the technique of using telescopes or dependent path types to encode heterogeneity is akin to finding the right interface for the algebraic structure of a type. But again, this does not scale to automation, and this limitation is independent of whether we are in a univalent type theory or not.

However using univalence *does* solve the computation issue of parametricity, as the function `diff` can be transported as a black box, by simply using an equality $\mathsf{e'}_{\mathbb{N}\mathsf{Bin}}$ between (ℕ; (0,S)) and (Bin; (0_{Bin},S_{Bin})) at type Pack := Σ A : Type, A * (A → A).

```
Definition diff_Bin : ∀ (n:Bin) (e : 0_Bin = S_Bin n), False :=
  eq_rect Pack (ℕ; (0,S)) (fun X _ ⇒ ∀ (n:X.1) (e : fst X.2 = snd X.2 n), False)
          diff (Bin; (0_Bin,S_Bin)) e'_ℕBin.
```

Therefore, it seems that a combination of parametricity and univalence could address all the issues identified thus far.

4 UNIVALENT PARAMETRICITY IN ACTION

This article develops the notion of *univalent parametricity* as a fruitful marriage of univalence and parametricity, which leverages their strengths while overcoming their limitations when taken in isolation. Specifically, univalent parametricity solves the anticipation problem of parametricity by using (a variant of) the heterogeneous parametricity translation, and solves the computation problem of parametricity by using univalence.

Given two equivalent types, univalent parametricity can be used to automatically transport properties defined on one type—*e.g.,* an easy-to-reason-about representation such as ℕ—to their counterparts on the other type—*e.g.,* a computationally-efficient representation such as Bin. Univalent parametricity provides the best of both parametricity and univalence in that, to transport a term, we can use either the White Box FP (Th 2.2) or the Black Box FP (Th 3.4), depending on the situation. In fact, the interplay between both modes is subtle in that white-box transport can automatically build univalent relations between two arbitrarily complex types based on user-provided relations, thereby inducing an equivalence that provides black-box transport for their inhabitants.

Univalent parametricity is a variant of the heterogeneous parametricity translation $a \approx_p b : A \bowtie_p B$ introduced in §2, simply noted $a \approx b : A \bowtie B$. When $a \approx b : A \bowtie B$ is inhabited, we say that a and b are *univalently related*. We sometimes omit A and B when they are clear from context.

The full development of univalent parametricity is in the following sections. In this section, we briefly illustrate univalent parametricity in action with programs and proofs over ℕ and Bin. First, we illustrate transport à la carte, *i.e.,* the possibility to refine automatic transport by establishing additional univalent relations (§4.1). Second, we show that univalent parametricity allows us to transport properties proven on ℕ to properties proven on Bin automatically (§4.2), and vice versa (§4.3).

4.1 Automatic Transport à la Carte

Having proven the type equivalence between $\mathbb{N}$ and Bin, we can prove that they are univalently related, *i.e.,* $\mathbb{N} \approx \mathsf{Bin} : \mathsf{Type} \bowtie \mathsf{Type}$. Doing so induces an automatic transport function, corresponding to the Black Box FP of univalence (Th 3.4). For instance we can transport a `square` function on $\mathbb{N}$ to an equivalent function on Bin:[5]

```
Definition square (x : ℕ) : ℕ := x * x.
Definition square_Bin■ : Bin → Bin := ↑■ square.
```

Note that from this section on, in code examples we use a general black-box transport operator $\uparrow_{\blacksquare}$ whose source and target types are inferred from context, and whose underlying equivalence is computed automatically, as will be explained in §7.

While $\mathtt{square}_{\mathsf{Bin}\blacksquare}$ is an effective function that can be used to compute the square of any binary natural number, it is inherently inefficient computationally, because of the black-box nature of the transport: when applied, $\mathtt{square}_{\mathsf{Bin}\blacksquare}$ first converts its Bin argument to an equivalent $\mathbb{N}$, applies the (slow) multiplication operation on $\mathbb{N}$, and finally converts back the $\mathbb{N}$ result to a Bin:

```
Check eq_refl : square_Bin■ = (fun x:Bin ⇒ ↑■ (square (↑■ x))).
```

At the cost of an additional proof effort, it is possible to establish that `mult` and $\mathtt{mult}_{\mathsf{Bin}}$ are univalently related, $\mathtt{mult} \approx \mathtt{mult}_{\mathsf{Bin}} : \mathbb{N} \to \mathbb{N} \bowtie \mathsf{Bin} \to \mathsf{Bin}$ (and likewise for `plus` and $\mathtt{plus}_{\mathsf{Bin}}$):

```
Definition univrel_mult : mult ≈ mult_Bin.
```

A first pay-off for this additional proof effort is that transport can now automatically exploit such a relation, so that we can transport `square` using the White Box FP of univalent parametricity, *i.e.,* rewriting its body and exploiting the univalent relation between `mult` and $\mathtt{mult}_{\mathsf{Bin}}$:

```
Definition square_Bin□ : Bin → Bin := ↑□ square.
```

The notation $\uparrow_{\square}$ corresponds to the $[\cdot]^{\Xi}_{\square}$ notation of the White Box FP (Th 2.2), where the global context Ξ is implicit. The management of the global context and the definition of this (ad hoc) polymorphic operator are realized in Coq through typeclasses, as will also be explained in §7.

The transported function $\mathtt{square}_{\mathsf{Bin}\square}$ now computes directly on Bin, instead of converting back and forth and using the (slow) multiplication operation on $\mathbb{N}$.

```
Check eq_refl : square_Bin□ = (fun x ⇒ (x * x)%Bin).
```

4.2 Automatic Transport of Properties

Establishing that two terms like `plus` and $\mathtt{plus}_{\mathsf{Bin}}$ are univalently related is not only valuable from a computational point of view. It also enables the automatic transport of properties that involve such terms.

Without presenting the details of univalent parametricity yet, suffice it to say that the type $\mathtt{plus} \approx \mathtt{plus}_{\mathsf{Bin}}$ actually unfolds to

[5] In the following, arithmetic operations in expressions are denoted with the same infix symbols (such as + and *); the actual operation is unambiguously determined by the type of its operands.

```
∀ (x : ℕ) (y : Bin), x = ↑■ y → ∀ (x' : ℕ) (y' : Bin), x' = ↑■ y' → x + x' = ↑■ (y + y')
```

which gives an extensional interpretation of the heterogeneous equality between `plus` and $\texttt{plus}_{\texttt{Bin}}$, using univalent transport $\uparrow_{\blacksquare}$ on terms of type Bin.

Then, thanks to the univalent relation $\texttt{plus} \approx \texttt{plus}_{\texttt{Bin}}$, it is possible to automatically infer the type equivalence between the type $\forall\ n\ m : \mathbb{N}, n + m = m + n$ and the type $\forall\ n\ m : \text{Bin}, n + m = m + n$. Consequently, the *proof* of commutativity for `plus` can automatically be transported to a proof of commutativity for $\texttt{plus}_{\texttt{Bin}}$:

```
Definition plus_Bin_comm : ∀ n m : Bin, n + m = m + n := ↑■ plus_comm.
```

Note that here, we do not face the computation issue encountered by using only parametricity (§2.2) because the term `plus_comm` is transported as a black box, *i.e.,* without recursively diving into its syntax.

In the same way, we can define the power function on both $\mathbb{N}$ and Bin and show that they are univalently related. Then, the following very simple proof of an additive property of the power function:

```
Definition pow_prop : ∀ n:ℕ, 3 ^ (n + 1)  = 3 * 3 ^ n.
 intro n; rewrite plus_comm; reflexivity.
Qed.
```

can be transported automatically to the power function on binary natural numbers:

```
Definition pow_Bin_prop : ∀ n:Bin, 3 ^ (n + 1) = 3 * 3 ^ n := ↑■ pow_prop.
```

In contrast, because adding 1 to a binary natural number is not an operation that preserves the inductive structure of that number, a direct proof of this lemma by induction on the binary natural number is much more involved.

4.3 Automatically Computing in the Equivalent Representation

Univalent parametricity can also be used the other way around to prove properties by computation on a type representation that is not always effective. Consider for instance the definition of a polynomial on natural numbers

```
Definition poly : ℕ → ℕ := fun n ⇒ 12 * n + 51 * n ^ 4 - n ^ 5.
```

and consider proving that `poly 50` is bigger than some given value, say 1000.[6] One would like to prove this by computation, *i.e.,* by actually calculating the value of `poly 50` and then simply comparing the result with 1000. However, because the unary representation is very inefficient, evaluating `poly` at 50 already exceeds the stack capacity of the Coq runtime.

```
Eval compute in poly 50.
Error Stack overflow
```

[6] We thank Assia Mahboubi for suggesting this example, taken from an actual mechanized mathematics exercise.

Therefore, the proof that poly 50 is bigger than 1000 cannot be done by computation. Univalent parametricity can overcome this issue by transporting the inequality to be proven to an equivalent one that uses the binary number representation.

```
Goal poly 50 ≥ 1000.
 replace_goal; now compute.
Defined.
```

The tactic replace_goal automatically infers, from poly 50 ≥ 1000, the univalently-related proposition on binary natural numbers using the White Box FP. Once the goal has been transported to a property on binary natural numbers, it is possible to proceed by computation, which produces a goal that can be solved automatically.

Note that automatic transport also works if we consider a slightly more complex example, where polynomials are encoded as a list of natural numbers representing its coefficients, together with a recursive evaluation function evalPoly:

```
Definition polyType := list ℕ.

Fixpoint evalPoly (p : polyType) (n : ℕ) (degree : ℕ) : ℕ :=
 match p with
 | [] ⇒ 0
 | coef :: p ⇒ coef * n ^ degree + evalPoly p n (S degree)
 end.

Infix "@@" := (fun p n ⇒ evalPoly p n 0).
```

Then, defining the polynomial poly in this setting and evaluating it at 50 leads to the exact same issue.

```
Definition poly' : polyType := [0;12;0;0;51;1].

Eval compute in poly' @@ 50.
Error Stack overflow
```

And the univalent parametricity framework allows again to transfer the goal to binary numbers in order to solve it by computation.

```
Goal poly' @@ 50 ≥ 1000.
 replace_goal; now compute.
Defined.
```

Because univalent parametricity is defined on all of CIC, this proof technique also scales to the definitions of fixpoints. For instance, consider the following sequence definition:

```
Fixpoint sequence (acc n : ℕ) :=
 match n with
```

```
  0 ⇒ acc
| 1 ⇒ 2 * acc
| 2 ⇒ 3 * acc
| S n ⇒ (sequence acc n) ^ acc
end.
```

Indeed, one can generically show that fixpoints preserve univalently-related arguments, which means that sequences producing unary natural numbers can be transported automatically to equivalent sequences producing binary natural numbers.

```
Definition sequence_prop : sequence 2 5 ≥ 1000.
 replace_goal; now compute.
Defined.
```

In summary, univalent parametricity follows the structural, white-box approach of parametricity to infer new univalent relations from existing ones, and can then exploit the induced equivalences as computational black boxes, as in univalence, to transport proofs and terms. The following sections develop the theory of univalent parametricity for CC_ω (§5) and CIC (§6), and its realization in the Coq proof assistant (§7). Then, we revisit the examples of this section, explaining how each step is implemented (§8), and discuss a case study for integrating native datatypes in Coq (§9).

5 UNIVALENT PARAMETRICITY

We now turn to the formal development of univalent parametricity. In this section, we focus on CC_ω—extension to CIC is in §6. We consider a type theory with the minimum requirements, namely the Calculus of Constructions with universes and the univalence axiom.

We first discuss in Section 5.1 the proper way to extend the parametricity translation in the universe to take equivalences into account. We then provide the complete univalent translation for CC_ω and present the Abstraction Theorem and its proof (§5.2). Note that the development here is largely independent of any particular realization of univalence, whether axiomatic or computational. In our axiomatic setting, let us insist that sometimes, a direct use of the univalence axiom would trivialize a proof but it would also suppress its computational content, thus leading to a useless framework in practice. In a setting where univalence is fully realized, such as in Cubical Type Theory, those simpler proofs could be used (see also §10).

In § 5.3, we extend the translation by taking a global context as input, which allows us to formulate the *White Box Fundamental Property* for CC_ω (Corollary 5.3). This property can relate terms of completely different types, such as inductively-defined and binary-encoded naturals, providing the global environment provides witnesses that they are in univalent relation. This is important because we want to be able to let programmers define their own equivalences and thus get univalent transport *à la carte* (§5.4). Univalent parametricity on types also entails type equivalence, which allows us to also state a *Black Box Fundamental Property* (Proposition 5.4), which says that when two types A and B are in univalent relation, any *open* term $t : A$ is in univalent relation with its (black-box) transport of type B.

5.1 Univalent Parametricity on the Universe

To strengthen parametricity to deal with equivalences, the univalent parametricity translation $[\![-]\!]_u$ must strengthen the parametricity translation on the universe Type_i. Several intuitive solutions come to mind, which however are not satisfactory.

First, we could simply replace the relation demanded by parametricity to be type equivalence itself, *i.e.,* $[\![\mathsf{Type}_i]\!]_u\ A\ B \triangleq A \simeq B$. However, by doing so, the abstraction theorem fails on $\vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}$. We would need to establish the fixpoint on the universe, *i.e.,* $[\![\mathsf{Type}_i]\!]_u : [\![\mathsf{Type}_{i+1}]\!]_u\ \mathsf{Type}_i\ \mathsf{Type}_i$, but we have

$$[\![\mathsf{Type}_i]\!]_u : \mathsf{Type}_i \to \mathsf{Type}_i \to \mathsf{Type}_{i+1} \neq \mathsf{Type}_i \simeq \mathsf{Type}_i.$$

In words, on the left-hand side we have an arbitrary relation on Type_i, while on the right-hand side, we have an equivalence.

Another intuitive approach is to state that $[\![\mathsf{Type}_i]\!]_u\ A\ B$ requires *both* a relation on A and B *and an equivalence* between A and B. While this goes in the right direction, it is insufficient because there is no connection between the two notions. This in particular implies that, when scaling up from CC_ω to CIC, the identity type—which defines the notion of equality—will not satisfy the abstraction theorem of univalent parametricity. We need to additionally demand that the relation *coincides with propositional equality* once the values are at the same type.

Therefore, an inhabitant of $[\![\mathsf{Type}_i]\!]_u$ is given by a relation $R : A \to B \to \mathsf{Type}_i$ and an equivalence $e : A \simeq B$, together with a *coherence condition* between the relation and the equivalence.[7] This (crucial!) condition stipulates that the relation does coincide with propositional equality up to a transport using the equivalence, *i.e.,* for all $a : A$ and $b : B$, the following should hold:[8]

$$R\ a\ b \simeq (a = \uparrow_e b)$$

This coherence condition allows us to show that once the relation is fixed, the rest of the data is a mere proposition. That is, the fact that there exists an equivalence satisfying the coherence condition just characterizes the kind of relations that can be used, but it does not provide additional structure. This way, univalent parametricity is really just a restriction of parametricity to relations that correspond to equivalences.

Therefore, for Type_i, we want the translation to be:

$$[\![\mathsf{Type}_i]\!]_u\ A\ B \triangleq \Sigma(R : A \to B \to \mathsf{Type}_i)(e : A \simeq B).\ \Pi a\ b.(R\ a\ b) \simeq (a = \uparrow_e b)$$

That is, the translation of a type (when seen as a term) needs to include a relation plus the fact that there is an equivalence, and that the relation is coherent with equality.

We therefore need to distinguish between the translation of a type T occurring in a *term position* (*i.e.,* left of the ":"), translated as $[T]_u$ and the translation of a type T occurring in a *type position* (*i.e.,* right of the ":"), translated as $[\![T]\!]_u$.[9] The abstraction theorem on Type_i enforces the definition

[7] Such an inhabitant is thus a dependent 3-tuple. We will later use syntactic sugar $t = (a; b; c)$ with accessors $t.1$ $t.2$ and $t.3$ for nested pairs to ease reading.

[8] From an equivalence $e : A \simeq B$ we can extract functions $\uparrow_B : A \to B$ and $\uparrow_A : B \to A$. We simply write $\uparrow_e$ to refer to either one as required by the context; these correspond to univalent (aka. black-box) transport. Also, in the following, when e is clear from the context, we often omit it and simply write $\uparrow$, as in the Coq examples of §4.

[9] The possibility to distinguish the translation of a type on the left and right-hand side of a judgment has already been noticed for other translations that add extra information to types by Boulier et al. [2017]. For instance, to prove the independence of univalence with CIC, they use a translation that associates a Boolean to any type, *e.g.,* $[\mathsf{Type}_i] = (\mathsf{Type}_i \times \mathbb{B}, \mathsf{true})$. Then a type on the left-hand side is translated as a 2-tuple and $[\![A]\!] = [A].1$. This possibility to add additional information in the translation of a type comes from the fact that types in CIC can only be "observed" through inhabitance, that is, in a type position; therefore, the translation in term positions may collect additional information.

of the translation on the universe hierarchy to satisfy:[10]

$$[\mathsf{Type}_i]_u : [\![\mathsf{Type}_{i+1}]\!]_u\ \mathsf{Type}_i\ \mathsf{Type}_i \quad \equiv$$
$$\Sigma(R : \mathsf{Type}_i \to \mathsf{Type}_i \to \mathsf{Type}_{i+1})(e : \mathsf{Type}_i \simeq \mathsf{Type}_i).\ \Pi a\ b.(R\ a\ b) \simeq (a =\uparrow_e b).$$

That is, $[\mathsf{Type}_i]_u$ must be itself a triple given by

$$[\mathsf{Type}_i]_u \triangleq (\lambda\ (A\ B : \mathsf{Type}_i),\ \Sigma(R : A \to B \to \mathsf{Type}_i)(e : A \simeq B).$$
$$\Pi ab.(R\ a\ b) \simeq (a =\uparrow_e b); \mathrm{id}_{\mathsf{Type}_i}; \mathrm{univ}_{\mathsf{Type}_i})$$

where $\mathrm{id}_{\mathsf{Type}_i}$ is simply the identity equivalence on the universe and $\mathrm{univ}_{\mathsf{Type}_i}$ is a proof that the univalent relation in the universe is coherent with equality on the universe as given below.

Proposition 5.1. *There exists a term* $univ_{\mathsf{Type}_i} : \Pi A\ B.\ [\![\mathsf{Type}_i]\!]_u\ A\ B \simeq (A = B)$.

Proof. The definition of $\mathrm{univ}_{\mathsf{Type}_i}$ crucially relies on univalence (and actually is equivalent to it), so this equivalence is not effective.

This result requires functional extensionality, *i.e.*, the fact that the canonical map

$$f = g \to \Pi(x : A).f\ x = g\ x$$

is an equivalence. This property is in fact a consequence of univalence [Univalent Foundations Program 2013].

By univalence and rearrangement of dependent sums, the type $[\![\mathsf{Type}_i]\!]_u\ A\ B$ is equivalent to

$$\Sigma(e : \mathsf{Type}_i \simeq \mathsf{Type}_i)(R : \mathsf{Type}_i \to \mathsf{Type}_i \to \mathsf{Type}_{i+1}).\ \Pi a\ b.R\ a\ b = (a =\uparrow_e b)$$

which by functional extensionality is equivalent to

$$\Sigma(e : \mathsf{Type}_i \simeq \mathsf{Type}_i)(R : \mathsf{Type}_i \to \mathsf{Type}_i \to \mathsf{Type}_{i+1}).\ R = (\lambda\ a\ b.\ a =\uparrow_e b)$$

But $\Sigma(R : \mathsf{Type}_i \to \mathsf{Type}_i \to \mathsf{Type}_{i+1}).\ R = (\lambda\ a\ b.\ a =\uparrow_e b)$ is a singleton type, which is always contractible. So we get an equivalence

$$[\![\mathsf{Type}_i]\!]_u\ A\ B \simeq (A \simeq B)$$

and we conclude by univalence again. □

5.2 The Univalent Parametricity Translation

We now turn to the full definition of the univalent parametricity translation on the whole syntax of CC_ω, including variables, application and lambda expressions, as a variation on the parametricity translation in the style of Bernardy *et al.* (recall Figure 2 of §2.1). Figure 4 shows how to extend the parametricity translation to force the relation defined between two types to correspond to a type equivalence with the coherence condition, as exposed in §5.1. Note that the translation does not target CC_ω but rather CIC_u, which is CIC augmented with the univalence axiom. We write $\Gamma \vdash_u t : T$ to stipulate that the term is typeable in CIC_u.

As explained in the previous section, the definition of the translation of a type A is more complex than that of Figure 2 because in addition to the relation $[\![A]\!]_u$, we need an equivalence $[\![A]\!]_u^{eq}$ and a witness $[\![A]\!]_u^{coh}$ that the relation is coherent with equality. This is why the translation of dependent products makes use of two additional terms Equiv_Π and univ_Π that will be explained during the proof of the Abstraction Theorem.

For the other terms, the translation does not change with respect to parametricity except that $[\![-]\!]_u$ must be used accordingly when we are denoting the relation induced by the translation and not the translation itself.

[10] $\equiv$ denotes equality by conversion.

$$
\begin{aligned}
[\mathsf{Type}_i]_u &\triangleq (\lambda\ (A\ B : \mathsf{Type}_i),\ \Sigma(R : A \to B \to \mathsf{Type}_i)(e : A \simeq B).\\
&\qquad \Pi ab.(R\ a\ b) \simeq (a =\uparrow_e b); \mathrm{id}_{\mathsf{Type}_i}; \mathsf{univ}_{\mathsf{Type}_i})\\
[\Pi a : A.\ B]_u &\triangleq (\lambda\ (f : \Pi a : A.B)\ (g : \Pi a' : A'.B').\\
&\qquad \Pi(a : A)(a' : A')(a^\varepsilon : [\![A]\!]_u\ a\ a').\ [\![B]\!]_u\ (f\ a)\ (g\ a');\\
&\qquad \mathrm{Equiv}_\Pi\ A\ A'\ [A]_u\ B\ B'\ [B]_u; \mathsf{univ}_\Pi\ A\ A'\ [A]_u\ B\ B'\ [B]_u)\\
[x]_u &\triangleq x^\varepsilon\\
[\lambda x : A.t]_u &\triangleq \lambda(x : A)(x' : A')(x^\varepsilon : [\![A]\!]_u\ x\ x').\ [t]_u\\
[t\ u]_u &\triangleq [t]_u\ u\ u'\ [u]_u\\
\\
[\![A]\!]_u &\triangleq [A]_u.1 \quad [\![A]\!]_u^{eq} \triangleq [A]_u.2 \quad [\![A]\!]_u^{coh} \triangleq [A]_u.3\\
\\
[\![\cdot]\!]_u &\triangleq \cdot\\
[\![\Gamma, x : A]\!]_u &\triangleq [\![\Gamma]\!]_u, x : A,\ x' : A',\ x^\varepsilon : [\![A]\!]_u\ x\ x'
\end{aligned}
$$

Fig. 4. Univalent parametricity translation for CC_ω

Theorem 5.2 (Abstraction theorem). *If* $\Gamma \vdash t : A$ *then* $[\![\Gamma]\!]_u \vdash_u [t]_u : [\![A]\!]_u\ t\ t'$.

Proof. The proof is a straightforward induction on the typing derivation. The only cases that differ from Theorem 2.1 are the typing rules for the universe and for the type of dependent functions. The case of the universe has already been addressed in § 5.1. For dependent products, we first need to provide a term that witnesses the fact that the dependent product is congruent with respect to equivalences in the following sense:

$$
\begin{aligned}
\mathrm{Equiv}_\Pi : \Pi\ &(A\ B : \mathsf{Type}_i)\ (UR_{AB} : [\![\mathsf{Type}_i]\!]_u\ A\ B).\\
&\Pi\ (P : A \to \mathsf{Type}_i)\ (Q : B \to \mathsf{Type}_i)\\
&\quad (UR_{PQ} : \Pi(a : A)\ (b : B).\ UR_{AB}.1\ a\ b \to [\![\mathsf{Type}_i]\!]_u\ (Pa)\ (Qb))).\\
&\quad (\Pi(a : A).\ P\ a) \simeq (\Pi(b : B).\ Q\ b)
\end{aligned}
$$

In particular, we have $UR_{AB}.2 : A \simeq B$ and $\lambda\ a\ b\ r,\ (UR_{PQ}\ a\ b\ r).2 : \Pi(a : A)\ (b : B).\ UR_{AB}.1\ a\ b \to P\ a \simeq Q\ b$. Using the coherence condition $UR_{AB}.3$ between $UR_{AB}.1\ a\ b$ and $a =\uparrow b$, this boils down to $\Pi(a : A).\ P\ a \simeq Q\ (\uparrow a)$.

At this point we can apply a standard result of HoTT, namely `equiv_functor_∀` in the Coq HoTT library [Bauer et al. 2017]. This lemma requires functional extensionality in the proof that the two transport functions form an equivalence.[11]

[11] The definition of the inverse function requires using the retraction, and the proof that it forms a proper equivalence requires the adjunction condition (§3.1). This means that the dependent function type would not be univalent if we replaced type equivalence with a simpler notion, such as the possibility to go from one type to another and back, or even by isomorphisms.

The second step is to provide a proof that the univalent (pointwise) relation on the dependent product is coherent with equality up to the equivalence above; *i.e.,* we need to define a term

$$\begin{aligned}&\mathsf{univ}_\Pi : \Pi\ (A\ B : \mathsf{Type}_i)\ (UR_{AB} : [\![\mathsf{Type}_i]\!]_u\ A\ B).\\ &\quad \Pi\ (P : A \to \mathsf{Type}_i)\ (Q : B \to \mathsf{Type}_i)\\ &\quad\ (UR_{PQ} : \Pi(a : A)\ (b : B).\ UR_{AB}.1\ a\ b \to [\![\mathsf{Type}_i]\!]_u\ (P\ a)\ (Q\ b))).\\ &\quad\ \Pi f\ g.(\Pi(a : A)(b : B)(r : UR_{AB}.1\ a\ b).\ (UR_{PQ}\ a\ b\ r).1\ (f\ a)\ (g\ a')) \simeq (f = \uparrow g)\end{aligned}$$

This part is quite involved. In essence, this is where we prove that transporting in many hard-to-predict places is equivalent to transporting only at the top level. This is done by repeated use of commutativity lemmas of transport of equality over functions. We refer the reader to the Coq development for more details. □

5.2.1 Prop. The definition of the univalent parametricity translation of Figure 4 does not deal with the universe Prop of proposition, but it can be treated in the same way as Type_i because Prop : Type_i is a universe that also enjoys the univalence axiom. The only specificity of Prop is its impredicativity, which does not play any role here.

It is also possible to state a stronger axiom on Prop called *propositional extensionality*, which uses logical equivalences instead of type equivalences in its statement:

$$(P = Q) \simeq (P \iff Q).$$

This axiom cannot be deduced from univalence alone, one would need proof irrelevance for Prop as well. As we are looking for the minimal amount of axioms needed for establishing univalent parametricity, we do not make use of this stronger axiom.

Note that exploiting the fact that Prop is proof irrelevant, $[\![\text{Prop}]\!]_u\ P\ Q$ boils down to

$$\Sigma(R : P \to Q \to \mathsf{Type}).\ (P \iff Q) * (\Pi(p : P)\ (q : Q).\ \mathtt{IsContr}(R\ p\ q)).$$

where $*$ is the product of types and $\mathtt{IsContr}\ A$ says that A is contractible, *i.e.,* it has a unique inhabitant. This is because for all p and q, the type $(p = \uparrow q)$ is contractible, and being equivalent to a contractible type is the same as being contractible. The definition we obtain in this case coincides with the definition of parametricity with uniformity of propositions developed by Anand and Morrisett [2017] (more details in §10).

5.3 White Box and Black Box Fundamental Properties

The extension of the parametricity translation that takes a global context into account presented in order to handle heterogeneous instance (§2.3) can also be applied to univalent parametricity. Recall that we consider a global context Ξ to be the following telescope Ξ_n defined as:

$$\begin{aligned}\Xi_0 &= \cdot\\ \Xi_1 &= (c_1^\circ : A_1^\circ;\ c_1^\bullet : A_1^\bullet;\ c_1^\otimes : [A_1]_u^{\Xi_0}\ c_1^\circ\ c_1{}^\bullet)\\ &\dots\\ \Xi_n &= \Xi_{n-1}, (c_n^\circ : A_n^\circ;\ c_n^\bullet : A_n^\bullet;\ c_n^\otimes : [A_n]_u^{\Xi_{n-1}}\ c_n^\circ\ c_n^\bullet)\end{aligned}$$

For simplicity, we reuse the notation of Section 2.3 to get a similar White Box FP for univalent parametricity. In particular, the definition of Figure 4 is likewise extended on constants as follows:

$$[c^\circ]_u^\Xi \triangleq c^\otimes \text{ when } (c^\circ : \_\ ;\ c^\bullet : \_\ ;\ c^\otimes : \_) \in \Xi$$

Corollary 5.3 (White Box Fundamental Property). *If* $|\Xi|_\circ \vdash a : A$ *then* $|\Xi|_\bullet \vdash [a]_\square^\Xi : [A]_\square^\Xi$ *and* $|\Xi| \vdash_u [a]_u^\Xi : [\![A]\!]_u^\Xi\ a\ [a]_\square^\Xi$.

Proof. Similar to the proof of Corollary 2.2. □

As we have done for parametricity, we introduce the notation

$$a \approx b : A \bowtie B \triangleq [\![A]\!]^{\Xi}_{u}\ a\ b \qquad (\text{when } B = [A]^{\Xi}_{\square})$$

which relates two terms a and b at two related—but potentially different—types A and B. This notation only makes sense when $B = [A]^{\Xi}_{\square}$ in the current global context Ξ, and we implicitly assume it is the case when we use this notation.

The White Box FP is the usual result obtained using parametricity, just rephrased using a context of global constants. But in the case of univalent parametricity, we have an additional property coming from the use of equivalences, which is *oblivious* to the structure of the term, and is defined for any well-typed terms, even in an open context, and in particular in presence of axioms. Indeed, once we know that two types A and B are univalently related, there is a canonical term in B related to any term t of type A obtained from the following property:

Proposition 5.4 (Black Box Fundamental Property). *Let A and B be two types such that $A \approx B : \mathsf{Type} \bowtie \mathsf{Type}$, then for all $t : A$, there is a proof that $t \approx \uparrow t : A \bowtie B$.*

Proof. By reflexivity of equality, $\uparrow t =_B \uparrow t$, which gives the desired result by using the coherence condition between equality and the relation. □

Note that the Black Box FP is internal to the theory, as opposed to the White Box FP, which is external. In particular, this means that the Black Box FP is valid in any context on open terms, which is crucial to define automatic transport.

5.4 Univalent Parametricity for Transport à la Carte

The Black Box Fundamental Property (Proposition 5.4) is a key advantage of univalent parametricity over traditional parametricity, because knowing that two types A and B are univalently related is enough to get a transport function from A to B, whereas traditional parametricity requires the exact definition of the term a in A in order to compute its counterpart in B. Now, it remains to investigate how to determine that two types are univalently related.

The White Box FP (Corollary 5.3) allows us to enrich the univalent relation "outside the diagonal", *i.e.,* to provide heterogeneous instances in the global context. For instance, we can relate unary naturals $\mathbb{N}$ and binary naturals Bin, *i.e.,* $\mathbb{N} \approx \mathsf{Bin} : \mathsf{Type} \bowtie \mathsf{Type}$. By combining this basic relation with the fact that type constructors are univalently parametric, it is possible to automatically derive that:

$$\mathbb{N} \to \mathbb{N} \to \mathbb{N} \approx \mathsf{Bin} \to \mathsf{Bin} \to \mathsf{Bin} : \mathsf{Type} \bowtie \mathsf{Type}$$

But more interestingly, not only univalent relation instances between types can be added, but also instances between any two terms, seen as new constants of the theory. For instance, consider the case of the definitions of the addition on unary and binary natural numbers $\mathtt{plus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $\mathtt{plus}_{\mathsf{Bin}} : \mathsf{Bin} \to \mathsf{Bin} \to \mathsf{Bin}$. One can show that they are univalently related

$$\mathtt{plus} \approx \mathtt{plus}_{\mathsf{Bin}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \bowtie \mathsf{Bin} \to \mathsf{Bin} \to \mathsf{Bin}$$

which, as illustrated in §4, allows automatic transport to be more computationally efficient, and proofs of results involving addition to be transported automatically. Indeed, there exists a context Ξ that relates

$$|\Xi|_{\circ} \equiv \mathbb{N}{:}\mathsf{Type}, \mathtt{plus}{:}\mathbb{N} \to \mathbb{N} \to \mathbb{N}, \mathtt{mult}{:}\mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

and

$$|\Xi|_{\bullet} \equiv \mathsf{Bin}{:}\mathsf{Type}, \mathtt{plus}_{\mathsf{Bin}}{:}\mathsf{Bin} \to \mathsf{Bin} \to \mathsf{Bin}, \mathtt{mult}_{\mathsf{Bin}}{:}\mathsf{Bin} \to \mathsf{Bin} \to \mathsf{Bin}.$$

As discussed in Section 4.1, applying the White Box FP to the term `square` directly gives us the term $\texttt{square}_{\texttt{Bin}\square}$ together with a proof that it is related to `square` in the context Ξ.

But the White Box FP also automatically gives us that the two following types are related

$$\forall\ (\texttt{n}\ \texttt{m} : \mathbb{N}),\ \texttt{plus}\ \texttt{n}\ \texttt{m} = \texttt{plus}\ \texttt{m}\ \texttt{n} \approx \forall\ (\texttt{n}\ \texttt{m} : \texttt{Bin}),\ \texttt{plus}_{\texttt{Bin}}\ \texttt{n}\ \texttt{m} = \texttt{plus}_{\texttt{Bin}}\ \texttt{m}\ \texttt{n} : \texttt{Type} \bowtie \texttt{Type}$$

because $\forall\ (\texttt{n}\ \texttt{m} : \texttt{Bin}),\ \texttt{plus}_{\texttt{Bin}}\ \texttt{n}\ \texttt{m} = \texttt{plus}_{\texttt{Bin}}\ \texttt{m}\ \texttt{n}$ is the prime translation of $\forall\ (\texttt{n}\ \texttt{m} : \mathbb{N}),\ \texttt{plus}\ \texttt{n}\ \ \texttt{m} = \texttt{plus}\ \texttt{m}\ \texttt{n}$. Then, using the Black Box FP provides a direct transport $\texttt{plus}_{\texttt{Bin}}\texttt{\_comm}$ of the proof of commutativity of addition `plus_comm`. Note that the computational content of `plus_comm` is not used to derive $\texttt{plus}_{\texttt{Bin}}\texttt{\_comm}$. Therefore, univalent parametricity naturally provides a framework to transport functions and proofs of theorems à la carte, depending on the univalent relation context that has been specified by the user, thanks to the interplay between white-box and black-box transports.

6 UNIVALENT PARAMETRICITY FOR INDUCTIVE TYPES

We now turn to the extension of univalent parametricity in theories that provide inductive types, such as the Calculus of Inductive Constructions (CIC) [Paulin-Mohring 2015]. We first give the general idea of the approach, and then proceed step-by-step, first considering dependent pairs (§6.1), records (§6.2), parameterized recursive inductive families (§6.3), and finally indexed inductive types (§6.4).

An inductive type is defined as a new type constructor, together with associated constructors and an elimination principle.[12] For instance, the inductive type of lists is[13]

```
Inductive list (A : Type) : Type :=
  nil : list A
| cons : A → list A → list A
```

where `nil` and `cons` are the constructors of the inductive type. The associated eliminator is

```
list_rect : ∀ (A : Type) (P : list A → Type), P nil →  (∀ (a : A) (l : list A), P l → P (a :: l))
          → ∀ l : list A, P l.
```

Let us consider an inductive type $\texttt{I} : \texttt{T}$, with constructors $\texttt{I\_c}_i : \texttt{T}_{\texttt{c}_i}$ and elimination principle $\texttt{I\_rect} : \texttt{T}_{\texttt{rect}}$. Using the global context presentation of Section 5.3, the introduction of this new inductive type in our framework amounts to the ability to extend the current global context Ξ with the following triple for the type constructor

$$(\texttt{I} : \texttt{T}; \texttt{I} : \texttt{T}; \texttt{I}^{\circledast} : [\![\texttt{T}]\!]_u^{\Xi}\ \texttt{I}\ \texttt{I})$$

and similar triples for each constructor and for the eliminator. We can then directly use the White Box FP (Corollary 5.3) extended with this inductive type. To sum up, the addition of the triples for the inductive type `I` amounts to giving the following terms[14]

$$\left[\begin{array}{lcl} \texttt{I}^{\circledast} & : & \texttt{I} \approx \texttt{I} : \texttt{T} \bowtie \texttt{T} \\ \texttt{I\_c}_i{}^{\circledast} & : & \texttt{I\_c}_i \approx \texttt{I\_c}_i : \texttt{T}_{\texttt{c}_i} \bowtie \texttt{T}_{\texttt{c}_i} \\ \texttt{I\_rect}^{\circledast} & : & \texttt{I\_rect} \approx \texttt{I\_rect} : \texttt{T}_{\texttt{rect}} \bowtie \texttt{T}_{\texttt{rect}} \end{array}\right.$$

[12] There is an equivalent presentation of inductive types with pattern matching instead of eliminators [Goguen et al. 2006]. In Coq, eliminators are automatically inferred and defined using pattern matching.

[13] In this section, to ease the reading, we navigate between the syntax of CIC and the one of Coq when appropriate.

[14] Here, and in the following, we use the notation $a \approx b : A \bowtie B$ extensively to avoid explicitly mentioning the global context.

When the above terms exist, we say that the inductive type, constructors and elimination principle are univalently parametric.

6.1 Dependent Pairs

In CIC, dependent pairs are defined as the inductive family:

```
Inductive sigT (A : Type) (B : A → Type) : Type :=
    existT : ∀ x : A, B x → sigT A B.
```

Thus, the unique constructor of a dependent pair is `existT` and the elimination principle is given by

```
sigT_rect : ∀ (A : Type) (P : A → Type) (P₀ : sigT A P → Type),
    (∀ (x : A) (p : P x), P₀ (x; p)) → ∀ s : sigT A P, P₀ s
```

As common, we use the notation $\Sigma a : A.\ B$ to denote `sigT A (fun a ⇒ B)`, similarly to dependent type theories where pair types are part of the syntax [Martin-Löf 1975].

Proposition 6.1. *There exists a term*

$$\Sigma^{\otimes} : \Sigma \approx \Sigma : \Pi(A : \mathsf{Type}_i).\ (A \to \mathsf{Type}_i) \to \mathsf{Type}_i \bowtie \Pi(A : \mathsf{Type}_i).\ (A \to \mathsf{Type}_i) \to \mathsf{Type}_i$$

Proof. The main steps of the construction are similar to those for the dependent product. Unfolding the definitions, giving a term $\Sigma^{\otimes}$ amounts to giving an inhabitant of $[\![\mathsf{Type}_i]\!]_u$ given two terms $UR_{AB} : A \approx B : \mathsf{Type}_i \bowtie \mathsf{Type}_i$ and $UR_{PQ} : P \approx Q : A \to \mathsf{Type}_i \bowtie B \to \mathsf{Type}_i$.

First, the univalent relation R_Σ between $\Sigma a : A.\ P\ a$ and $\Sigma b : B.\ Q\ b$ is defined as

$$R_\Sigma \triangleq \lambda(p : \Sigma a : A.\ P\ a)(q : \Sigma b : B.\ Q\ b).\ \Sigma(r_{pq} : UR_{AB}.1\ p.1\ q.1).\ (UR_{PQ}\ p.1\ q.1\ r_{pq}).1\ p.2\ q.2.$$

It naturally requires the first and second elements of the pair to be related at the corresponding types.

Second, the proof that $\Sigma a : A.\ P\ a \simeq \Sigma b : B.\ Q\ b$ also follows from a standard result of HoTT, namely `equiv_functor_sigma` in the Coq HoTT library. Contrarily to the dependent product, which requires functional extensionality, this lemma does not require any axiom.

Finally, the proof that the relation is coherent with equality is the novel part required by univalent parametricity. This means that we need to define a term:

$$\begin{aligned}&\mathsf{univ}_\Sigma : \Pi\ (A\ B : \mathsf{Type}_i)\ (UR_{AB} : A \approx B : \mathsf{Type}_i \bowtie \mathsf{Type}_i).\\ &\qquad \Pi\ (P : A \to \mathsf{Type}_i)\ (Q : B \to \mathsf{Type}_i)\ (UR_{PQ} : P \approx Q : A \to \mathsf{Type}_i \bowtie B \to \mathsf{Type}_i).\\ &\qquad\quad \Pi x\ y.(x \approx y : \Sigma a : A.\ P\ a \bowtie \Sigma b : B.\ Q\ b) \simeq (x = \uparrow y)\end{aligned}$$

Instead of building the equivalence explicitly with the transport functions and their associated section and retraction proofs, this equivalence can be conveniently proven by composition of equivalences. Specifically, we rely on a decomposition of equality for dependent sums:

$$(x \approx y) \equiv (\Sigma p : x.1 \approx y.1.\ x.2 \approx y.2) \simeq (\Sigma p : x.1 = \uparrow y.1\ .\ x.2 = \uparrow y.2) \simeq (x = \uparrow y)$$

Note that the last equivalence above is the counterpart of functional extensionality for dependent function types. The main difference is that this equivalence is effective as it can be proven by elimination of dependent pairs. □

With $\Sigma^{\otimes}$ defined, we can extend the global context for the constructor and eliminator as well.

Proposition 6.2. *There is a term*

$$\texttt{existT}^{\circledast} : \texttt{existT} \approx \texttt{existT} : \mathsf{T}_{\mathsf{ex}} \bowtie \mathsf{T}_{\mathsf{ex}}$$

where $\mathsf{T}_{\mathsf{ex}} \triangleq \Pi(A : \mathsf{Type})(P : A \to \mathsf{Type})(x : A).\ P\ x \to \Sigma(x : A).\ P\ x$ *and similarly for* `sigT_rect`.

Proof. Direct by induction on the structure of a dependent pair type. □

6.2 Record Types

The treatment of dependent pairs above scales to dependent records, by considering their encoding as iterated dependent pairs.

To illustrate, let us consider the example of a simple library record type `Lib`, which abstracts over an indexed container type constructor `C`, and packages functions `head` and `map` together with a property on their composition:

```
Record Lib (C : Type → ℕ → Type) :=
 { head : ∀ {A : Type} {n : ℕ}, C A (S n) → A;
   map : ∀ {A B} (f :A → B) {n}, C A n → C B n;
   prop : ∀ n A B (f : A → B) (v : C A (S n)), head (map f v) = f (head v)}.
```

Like all record types, `Lib` can be formulated in terms of nested dependent pairs. This means that, for any `C : Type → ℕ → Type`, `Lib C` is equivalent to

```
Lib' C := Σ (hd : ∀ A n. C A (S n) → A),
          Σ (map : ∀ A B (f:A → B) n, C A n → C B n),
            ∀ n A B (f : A → B) (v : C A (S n)), hd (map f v) = f (hd v).
```

The fact that `Lib'` is univalently parametric directly follows from the abstraction theorem of CC_ω extended with dependent pairs. To conclude that `Lib` is univalently parametric, we use the fact that a type family equivalent to a univalently parametric type family is itself univalently parametric.

This approach to establish the univalent parametricity record type via its encoding with dependent pairs can be extended to any record type. We have automatized this principle in our Coq framework as a tactic, by reusing an idea used in the HoTT library that allows automated inference of type equivalence for records with their nested pair types formulation. This tactic can be used to automatically prove that a given record type is univalently parametric (provided its fields are).

6.3 Parameterized Recursive Inductive Families

To establish the univalent parametricity of a *parameterless* recursive inductive type, such as natural numbers with zero and successor, we can simply use the canonical structure over the identity equivalence, with equality as univalent relation and trivial coherence. However, whenever an inductive type has parameters, the situation is more complex.[15]

Let us develop the case of lists and define the term

$$\texttt{list}^{\circledast} : \texttt{list} \approx \texttt{list} : \mathsf{Type}_i \to \mathsf{Type}_i \bowtie \mathsf{Type}_i \to \mathsf{Type}_i.$$

[15] In this work the distinction between *parameters* and *indices* for inductive types is important. A parameter is merely indicative that the type behaves *uniformly* with respect to the supplied argument. For instance A in `list` A is a parameter. Thus the choice of A only affects the type of elements inside the list, not its shape. In particular, by knowing A for a given list, we cannot infer which constructor was used to construct the list. On the other hand, n in `Vect` A n is an index. By knowing the value of an index, one can infer which constructor(s) may or may not have been used to create the value. For instance, a value of type `Vect` A 0 is necessarily the empty vector. We address indexed inductive types in §6.4.

Unfolding the definitions, giving a term $\texttt{list}^{\circledast}$ amounts to exhibiting an inhabitant of $[\![\texttt{Type}_i]\!]_u$ given two types A and B related by $UR_{AB} : A \approx B : \texttt{Type}_i \bowtie \texttt{Type}_i$.

The univalent relation on lists is given directly by parametricity. Indeed, following the work of Bernardy et al. [2012] on the inductive-style translation, the inductive type corresponding to the transport of a relation between A and B to a relation between list A and list B is given by:

```
Inductive UR_list A B (R : A → B → Type) : list A → list B → Type :=
 UR_list_nil : UR_list R nil nil
| UR_list_cons : ∀ a b l l', (R a b) → (UR_list R l l') → UR_list R (a::l) (b::l').
```

This definition captures the fact that two lists are related iff they are of the same length and pointwise-related. Then, the univalent relation is given by

$$R_{\texttt{list}} \triangleq \lambda(l : \texttt{list}\ A)(l' : \texttt{list}\ B).\ \texttt{UR\_list}\ A\ B\ UR_{AB}.1\ l\ l'$$

Then, we need to show that

$$\texttt{list}\ A \simeq \texttt{list}\ B$$

knowing that $A \simeq B$ from UR_{AB}. The two transport functions of the equivalence $\texttt{list}\ A \simeq \texttt{list}\ B$ can be defined by induction on the structure of the list (*i.e.,* using the eliminator list_rect). They both simply correspond to the usual map operation on lists. The proofs of the section and retraction are also direct by induction on the structure of the list, and transporting along the section and retraction of $A \simeq B$.

Similarly to dependent pairs, the proof that the relation is coherent with equality relies on the following decomposition of equality between lists:

$$\Pi\ A\ B\ (e : A \simeq B)\ l\ l'.\ (\texttt{UR\_list}\ A\ B\ (\lambda\ a\ b.a = \uparrow b)\ l\ l') \simeq (l = \uparrow l')$$

Indeed, using this lemma, the coherence of the univalent relation with equality is easy to infer:

$$(l \approx l') \equiv (\texttt{UR\_list}\ A\ B\ UR_{AB}.1\ l\ l') \simeq (\texttt{UR\_list}\ A\ B\ (\lambda\ a\ b.a = \uparrow b)\ l\ l') \simeq (l = \uparrow l')$$

Note that it is always valid to decompose equality on inductive types. This is because a value of an inductive type can only be observed by analyzing which constructor was used to build the value. This fact is explicitly captured by the elimination principle of an inductive type. On the contrary, for dependent products, the fact that functions can only be observed through application to a term is implicit in CIC, *i.e.,* there is no corresponding elimination principle in the theory (hence functional extensionality is an axiom).

The proofs that the constructors nil and cons are univalently parametric are direct by definition of UR_List. Likewise, the proof that the eliminator list_rect is univalently parametric is direct by induction on UR_List.

Generalization. It is possible to generalize the above result, developed for lists, to any parameterized inductive family. As illustrated above, the univalent relation for parameterized inductive families is given by parametricity, and the proof that related inputs give rise to equivalent types proceeds by a direct induction on the structure of the type. The main difficulty is to generalize the proof of the coherence of the relation with equality. Indeed, this involves fairly technical reasoning on equality and injectivity of constructors.

Fortunately, in practice in our Coq framework, a general construction is not required to handle each new inductive type, because a witness of the fact that a given inductive is univalently

parametric can be defined specifically as a typeclass instance. We also provide a tactic to automatically generate this proof on any parameterized datatype (up to a fixed number of constructors), depending on the univalent parametricity of its parameters.

6.4 Indexed Inductive Families

CIC supports the definition of inductive types that are not only parameterized, but also indexed, like length-indexed vectors `Vector` A n. Another mainstream example is Generalized Algebraic Data Types (GADTs) [Peyton Jones et al. 2006] illustrated here with the typical application to modeling typed expressions:

```
Inductive Expr : Type → Type :=
| I : ℕ → Expr ℕ
| B : 𝔹 → Expr 𝔹
| Ad : Expr ℕ → Expr ℕ → Expr ℕ
| Eq : Expr ℕ → Expr ℕ → Expr 𝔹.
```

Observe that the return types of constructors instantiate the inductive family at specific type indices, instead of uniform type parameters as is the case for *e.g.,* the parameterized list inductive type. This specificity of constructors is exactly what makes GADTs interesting for certain applications; but this is precisely why their univalent parametricity is ineffective!

Indeed, consider the equivalence between natural numbers $\mathbb{N}$ and binary natural numbers `Bin`. Univalent parametricity of the `Expr` GADT means that `Expr` $\mathbb{N}$ is equivalent to `Expr Bin`. However, there is no constructor for `Expr` that can produce a value of type `Expr Bin`. So the only way to obtain such a term is by using an *equality* between $\mathbb{N}$ and `Bin`, that is, using the univalence axiom.[16]

The challenge is that univalent parametricity for indexed inductive families relies on the coherence condition. To better understand this point, let us study the prototypical case of identity types.

Identity types. In Coq, the identity (or equality) type `eq`, with notation `=`, is defined as an indexed inductive family with a single constructor `eq_refl`:

```
Inductive eq (A : Type) (x : A) : A → Type := eq_refl : x = x.
```

The elimination principle `eq_rect`, known as *path induction* in HoTT terminology, is:

```
eq_rect : ∀ A (x : A) (P : ∀ a : A, x = a → Type), P x eq_refl → ∀ (y : A) (e : x = y), P y e
```

Proposition 6.3. *There is a term*

$$\mathsf{eq}^{\circledast} : \mathsf{eq} \approx \mathsf{eq} : \Pi\ (A : \mathsf{Type})\ (x : A).\ A \to \mathsf{Type} \bowtie \Pi\ (A : \mathsf{Type})\ (x : A).\ A \to \mathsf{Type}.$$

Proof. Unfolding the definition, this amounts to defining an inhabitant of $[\![\mathsf{Type}]\!]_u$ given two types A, B related by $UR_{AB} : A \approx B : \mathsf{Type} \bowtie \mathsf{Type}$, two terms a in A, b in B related by $e : a \approx b : A \bowtie B$ and two terms a' in A', b' in B related by $e' : a' \approx b' : A \bowtie B$.

The univalent relation for identity types is defined using the inductive type obtained by applying parametricity to the identity type:

[16] It is however impossible to *prove* that no term of type `Expr Bin` can be constructed without univalence, because the univalence axiom is compatible with CIC.

```
Inductive UR_eq (A₁ A₂ : Type) (A_R : A₁ → A₂ → Type) (x₁ : A₁) (x₂ : A₂) (x_R : A_R x₁ x₂) :
  ∀ (y₁ : A₁) (y₂ : A₂), A_R y₁ y₂ → x₁ = y₁ → x₂ = y₂ → Type :=
  UR_eq_refl : UR_eq A₁ A₂ A_R x₁ x₂ x_R x₁ x₂ x_R eq_refl eq_refl.
```

The univalent relation is just a specialization of UR_eq where A_R is given by the relation induced by UR_{AB} and with e and e', so we set:

$$R_{\mathsf{eq}} \triangleq \lambda(e_1 : a =_A a')(e_2 : b =_B b').\ \mathtt{UR\_eq}\ A\ B\ UR_{AB}.1\ a\ b\ e\ a'\ b'\ e'\ e_1\ e_2$$

To prove that $(a = a') \simeq (b = b')$, it is necessary to use the coherence between the relation and the equivalence provided by UR_{AB}. After rewriting, the equivalence to establish is

$$(a =_A a') \simeq (\uparrow a =_B \uparrow a')$$

This equivalence is again similar to a standard result of HoTT, namely equiv_functor_eq in the Coq HoTT library.

Finally, proving that the relation is coherent with equality amounts to show that

$$\Pi(e_1\ e_2 : a = a').\ (e = e') \simeq \mathtt{UR\_eq}\ A\ B\ UR_{AB}.1\ a\ b\ e\ a'\ b'\ e'\ e_1\ \uparrow e_2$$

This can be done by first showing the following equivalence[17]

$$\mathtt{UR\_eq}\ A\ B\ P\ x\ y\ H\ x'\ y'\ H'\ X\ Y \simeq (Y\ \#\ (X\ \#\ H) = H')$$

which means that the naturality square between H and H' commutes. □

The proofs that eq_refl and eq_rect are univalently parametric are direct by UR_eq_refl and elimination of UR_eq.

To deal with other indexed inductive types, one can follow a similar approach. Alternatively, it is possible to exploit the correspondence between an indexed inductive family and a subset of parameterized inductive family, established by Gambino and Hyland [2004], to prove the univalence of an indexed inductive family. In this correspondence, the property of the subset type is obtained from the identity type.

For instance, for vectors:

$$\mathtt{Vector}\ A\ n \simeq \Sigma\ l : \mathtt{list}\ A.\ \mathtt{length}\ l = n$$

The length function computes the length of a list, as follows:

```
Definition length {A} (l: list A) : ℕ := list_rect A (fun _ ⇒ ℕ) O (fun _ l n ⇒ S n) l
```

where one can observe that the semantics of the index in the different constructors of vectors is captured in the use of the recursion principle list_rect. By the abstraction theorem, $\Sigma\ l : \mathtt{list}\ A.\ \mathtt{length}\ l = n$ is univalently parametric, and thus so is Vect $A\ n$.

[17] The notation $e\ \#\ t$, with $e : x = y$ and $t : P\ x$ when P is clear from the context, denotes the transport of the term e through the equality proof e (hence $e\ \#\ t : P\ y$).

7 UNIVALENT PARAMETRICITY IN COQ

The whole development of univalent parametricity exposed in this article has been formalized and implemented in the Coq proof assistant [Coq Development Team 2019], reusing several constructions from the HoTT library [Bauer et al. 2017]. We have formalized in Coq the univalent parametricity translation, in order to mechanically verify the content from §5—we do not discuss this effort here. Instead, we present the shallow embedding of the univalent relation in Coq, based on typeclass instances to define and automatically derive the univalent parametricity proofs of Coq constructions. This framework brings the benefits of univalent parametricity to standard Coq developments.

We first introduce the core classes of the framework (§7.1), and then describe the instances for some type constructors (§7.2). We explain how the use of typeclasses gives rise to a direct implementation of univalent transport à la carte (§7.3). Finally, we discuss several refinements to the framework to circumvent the limitation of relying on the univalence axiom in Coq (§7.4).

7.1 Coq Framework

The central notion at the heart of this work is that of type equivalences, which we formulate as a typeclass to allow automatic inference of equivalences:[18]

```
Class IsEquiv (A B : Type) (f : A → B) := {
 e_inv : B → A;
 e_sect : ∀ x, e_inv (f x) = x;
 e_retr : ∀ y, f (e_inv y) = y;
 e_adj  : ∀ x, e_retr (f x) = ap f (e_sect x) }.
```

The properties `e_sect` and `e_retr` express that `e_inv` is both the left and right inverse of `f`, respectively. The property `e_adj` is a compatibility condition between the proofs. It ensures that the equivalence is uniquely determined by the function `f`.

While `IsEquiv` characterizes a particular function f as being an equivalence, we say that two types A and B are equivalent, noted $A \simeq B$, iff there exists such a function f.

```
Class Equiv A B := { e_fun :> A → B ; e_isequiv : IsEquiv e_fun }.
Infix "≃" := Equiv.
```

`Equiv` is here defined as a typeclass to allow automatic inference of equivalences. This way, we can define automatic univalent transport as follows:

```
Definition univalent_transport {A B : Type} {e : A ≃ B} : A → B := e_fun e.
Notation "↑" := univalent_transport.
```

where the equivalence is obtained through typeclass instance resolution, *i.e.,* proof search. Note also that the source and target types of transport are implicitly resolved by default.

To formalize univalent relations, we define a hierarchy of classes, starting from `UR` for univalent relations (arbitrary heterogeneous relations), refined by `UR_Coh`, which additionally requires the proof of coherence between a univalent relation and equality.

[18] Adapted from: http://hott.github.io/HoTT/coqdoc-html/HoTT.Overture.html.

```
Class UR A B := { ur : A → B → Type }.
Infix "≈" := ur.

Class UR_Coh A B (e : Equiv A B) (H : UR A B) := { ur_coh : ∀ (a a' : A), Equiv (a = a') (a ≈ ↑■ a') }.
```

The attentive reader will notice that the definition of the coherence condition above is dual to the one stated in Figure 4. Both definitions are in fact equivalent.[19] The reason for adopting this dual definition is that it eases the definition of new instances.

As presented in Figure 4, two types are related by the univalent parametricity relation if they are equivalent and there is a coherent univalent relation between them. This is captured by the typeclass UR_Type.

```
Class UR_Type A B := {
  Ur :> UR A B;
  equiv :> A ≃ B;
  Ur_Coh :> UR_Coh A B equiv Ur;
  Ur_Can_A :> Canonical_eq A;
  Ur_Can_B :> Canonical_eq B }.
Infix "⋈" := UR_Type.
```

The last two attributes are part of the Coq framework in order to better support extensibility and effectiveness, as will be described in §7.4.1.

7.2 Univalent Type Constructors

The core of the development is devoted to the proofs that standard type constructors are univalently parametric, notably Type and Π. In terms of the Coq framework, this means providing UR_Type instances relating each constructor to itself. These instance definitions follow directly the proofs discussed in §5.

For the universe Type_i, we define:

```
Instance UR_Type_def@{i j} : UR@{j j j} Type@{i} Type@{i} := {| ur := UR_Type@{i i i} |}.
```

This is where our fixpoint construction appears: the relation at Type_i is defined to be UR_Type itself. So, for a type to be in the relation means more than mere equivalence: we also get a relation between elements of that type that is coherent with equality. This UR_Type_def instance will be used implicitly everywhere we use the notation X ≈ Y, when X and Y are types themselves.

For dependent function types, we set:

```
Definition UR_Forall A A' (B : A → Type) (B' : A' → Type) (dom : UR A A')
  (codom: ∀ x y (H : x ≈ y), UR (B x) (B' y)) : UR (∀ x, B x) (∀ y, B' y) :=
    {| ur := fun f g ⇒ ∀ x y (H: x ≈ y), f x ≈ g y |}.
```

[19] See lemma is_equiv_alt_ur_coh in the Coq development.

The univalent parametricity relation on dependent function types expects relations on the domain and codomain types, the latter being parameterized by the former through its argument (H : x ≈ y). The definition is the standard extensionality principle on dependent function types.

Then, we need to show that the universe is related to itself according to the relation on the universe at one level above. This corresponds to Proposition 5.1, and thanks to both universe polymorphism and implicit management of universe levels in Coq, there is no need to mention levels at all (note that Type ≈ Type in the definition actually computes to Type ⋈ Type):[20]

```
Definition FP_Type : Type ≈ Type.
```

Regarding the dependent product, the Equiv instance that needs to be defined in order to show that it is univalently parametric has the following type:

```
Instance Equiv_∀ : ∀ (A A' : Type) (e_A : A ≈ A') (B : A → Type)
  (B' : A' → Type) (e_B : B ≈ B'), (∀ x : A, B x) ≃ (∀ x : A', B' x).
```

While the conclusion is an equivalence, the assumptions e_A and e_B are about univalent relations for A, A' and B and B'. The first one is implicitly resolved as the UR_Type_def defined above, and the second one as a combination of UR_Forall and UR_Type_def. With these stronger assumptions, and because ≈ is heterogeneous, we can prove the equivalence without introducing transports. This is key to make the typeclass instance proof search tractable: it is basically structurally recursive on the type indices. We can then show that the dependent function type seen as a binary type constructor is related to itself using the univalent relation and equivalence constructed above, and the coherence proof univ_Π presented in the Abstraction Theorem (Th. 5.2):

```
Definition FP_∀ : (fun A B ⇒ (∀ x : A , B x)) ≈ (fun A' B' ⇒ (∀ x : A', B' x)).
```

To instrument the typeclass instance proof search, we add proof search Hints for each fundamental property.

We proceed similarly for other constructors: dependent pairs, the identity type, natural numbers and booleans with the canonical univalent relation, where we additionally prove the fundamental property for the eliminators. That is to say, we have many fundamental property lemmas such as:

```
Definition FP_Σ : @sigT ≈ @sigT.
```

Having spelled out the basics of the Coq framework for univalent parametricity, we can now turn to the practical issue of effective transport.

7.3 Univalent Transport à la Carte

Because the computation of the univalent parametricity relation is done using type class resolution, the framework is already set up to support extension with contexts of univalently-related constants, using typeclass instances. For example, to extend the context with the fact that two types A and B are univalently related, one needs to provide a proof that there is an equivalence between A and B, and declare it as a typeclass instance. Then, using for example the canonical relation fun (a : A) (b : B) ⇒ a = ↑■ b, one can construct an instance of A ⋈ B. The user also needs to define an instance of B ⋈ A using the symmetry of the relation, because typeclass resolution cannot be

[20] Some universe annotations appear in the Coq source files in order to explicitly validate our assumptions about universes.

instrumented with a general rule for symmetry, otherwise proof search would never terminate. This way, typeclass resolution is able to automatically derive further instances of the relation based on this univalent relation instance.

Note that the relation between A and B does not have to be the canonical relation. For instance, coming back to the example of the equivalence between sized lists and vectors, the relation between sized lists and vectors can be based either on equality (plus transport using the equivalence), on the relation UR_list (plus transport using the equivalence), on a similar relation on vectors (plus transport using the equivalence) or even on a heterogeneous relation directly relating sized lists and vectors (without the use of transport). Of course, all these definitions are equivalent because of the coherence condition of univalent parametricity, but they can have different computational content and a user can favor one or the other, depending on the application in mind.

Going one step further, one can show that two functions defined on univalently-related types are in univalent relation. Let us say for example that $\mathtt{f} : \mathtt{A} \to \mathtt{A} \to \mathtt{A}$ and $\mathtt{g} : \mathtt{B} \to \mathtt{B} \to \mathtt{B}$ are in univalent relation, with witness univrel_fg. Then, to exploit this univalent relation to perform univalent transport à la carte, the user needs to add the following Hint to the proof search database:[21]

```
Hint Extern 0 (f _ _ ≈ g _ _) ⇒ eapply univrel_fg : typeclass_instances.
```

This declaration amounts to extending the global context of univalently-related constants with f and g.

In general, when adding new instances to univalent parametricity, the user needs to define corresponding Hints to enable automatic typeclass resolution. §8 explains in more details how the examples of §4 are realized, including such hints.

7.4 Effectiveness of Univalent Parametric Transport in Coq

The proofs of univalent parametricity we have developed in §5 are in a setting where univalence is realized as an axiom (CC_ω and CIC). The axiomatic nature of the development manifests as follows:

(1) The univalence axiom proper is used to show the coherence condition of univalent parametricity for the universe (§5.1). This is to be expected and unavoidable, as this condition for the universe exactly states that type equivalence coincides with equality.
(2) Functional extensionality (an axiom in CIC, which follows from univalence) is used to show that the transport functions of the equivalence for the dependent product form an equivalence (§5.2).

Additionally, as shown in §6, the effectiveness of univalent transport for inductive types depends on the type of parameters and indices. In particular, proving univalent parametricity of indexed families requires using the coherence condition.

To see how this relates to practice, consider the case of functions. Functional extensionality is only used in the proof that the transport functions form an equivalence. In particular, this means that the transport functions themselves *are* effective. Therefore, when transporting a first-order function, the resulting function is effective.

For the axiom to interfere with effectiveness, we need to consider a *higher-order* function, *i.e.,* that takes another function as argument. Consider for instance the conversion of a higher-order dependent function g operating on a function over natural numbers

```
g : ∀ (f : ℕ → ℕ), Vector ℕ (f 0)
```

[21] We use hint declarations to have precise control over the shape of goals a typeclass instance should solve and how.

to one operating on a function over binary natural numbers

```
g' : ∀ (f : Bin → Bin), Vector Bin (f ↑■ 0) := ↑■ g.
```

We transport g to g' along the equivalence between the two higher-order types above. Such a transport uses, *in a computationally-relevant position*, the fact that the function argument f can be transported along the equivalence between $\mathbb{N} \to \mathbb{N}$ and Bin → Bin. Consequently, the use of functional extensionality in the equivalence proof chimes in, and g' is not effective. (Specifically, g' pattern matches on an equality between natural numbers that contains the functional extensionality axiom.)

Fortunately, there are different ways to circumvent this problem, by exploiting the fact that univalent parametricity is specializable through specific typeclass instances. This section shows how we can further specialize proofs of univalent parametricity in situations where using axioms can be avoided. Sometimes we can ignore the fact that an equality proof might be axiomatic by automatically crafting a new one that is axiom-free (§7.4.1), or we can avoid transporting type families with (potentially axiomatic) proofs of equality in some cases (§7.4.2).

7.4.1 Canonical Equality for Types with Decidable Equality. Any proof of equality between two natural numbers can be turned into a canonical, axiom-free proof using decidability of equality on natural numbers. In general, decidable equality on a type A can be expressed in type theory as

```
Definition DecEq (A : Type) := ∀ x y : A, (x = y) + ¬(x = y).
```

Hedberg's theorem [Hedberg 1998] implies that if A has decidable equality, then A satisfies Uniqueness of Identity Proofs (UIP): any two proofs of the same equality between elements of A are equal. Hedberg's theorem relies on the construction of a canonical equality to which every other is shown equal. Specifically, when A has a decidable equality, it is possible to define a function

```
Definition Canonical_eq_decidable A (Hdec : DecEq A) : ∀ x y : A, x = y → x = y :=
 fun x y e ⇒ match Hdec x y with
        | inl e0 ⇒ e0
        | inr n ⇒ match (n e) with end end.
```

This function produces an equality between two terms x and y of type A by using the decision procedure Hdec, independently of the equality e. In the first branch, when x and y are equal, it returns the canonical proof produced by Hdec, instead of propagating the input (possibly-axiomatic) proof e. And in case the decision procedure returns an inequality proof (of type x=y → False), the function uses e to establish the contradiction. In summary, the function transforms any equality into a canonical equality by using the input equality *only in cases that are not possible.*

We can take advantage of this insight to ensure effective transport on indices of types with decidable equality. The general idea is to extend the relation on types $A \bowtie B$ to also include two functions ∀ x y : A , x = y → x = y and ∀ x y : B , x = y → x = y. For types with decidable equality, these functions can exploit the technique presented above, and for others, these are just the identity. However, care must be taken: we cannot add arbitrary new computational content to the relation; we have to require that these functions preserve reflexivity. This is specified in the following class:

```
Class Canonical_eq (A : Type) :=
```

```
{ can_eq : ∀ (x y : A), x = y → x = y ;
  can_eq_refl : ∀ x, can_eq x x eq_refl = eq_refl }.
```

which is used for the last two attributes of the UR_Type class given in §7.1.

There are two canonical instances of Canonical_eq, the one that is defined on types with decidable equality, and exploits the technique above, and the default one, which is given by the identity function (and proof by reflexivity).

Using this extra information, it is possible to improve the definition of univalent parametricity by always working with canonical equalities. This way, equivalences for inductive types whose indices are of types with decidable equality—like length-indexed vectors and many common examples—never get stuck on rewriting of indices.

7.4.2 Canonically-Transportable Predicates. As mentioned in the introduction of this section, for some predicates, it is not necessary to pattern match on equality to implement transport.

The simplest example is when the predicate does not actually depend on the value, in which case P x ≃ P y can be implemented by the identity equivalence because P x is convertible to P y, independently of what x and y are. It is also the case when the predicate is defined on a type with a decidable equality, so we can instead pattern match on the canonical equality (§7.4.1).

To take advantage of this situation whenever possible, we introduce the notion of *transportable* predicates.

```
Class Transportable {A} (P : A → Type) := {
  transportable :> ∀ x y, x = y → P x ≃ P y;
  transportable_refl : ∀ x, transportable x x eq_refl = Equiv_id (P x) }.
```

Note that as for Canonical_eq, we need to require that transportable behaves like the standard transport of equality by sending reflexivity to the identity equivalence.

For instance, the instance for constant type-valued functions is defined as

```
Instance Transportable_cst A B : Transportable (fun _ : A ⇒ B) := {|
  transportable := fun (x y : A) _ ⇒ Equiv_id B;
  transportable_refl := fun x : A ⇒ eq_refl |}.
```

To propagate the information that every predicate (a.k.a. type family) comes with its instance of Transportable, we specialize the definition of UR (A → Type) (A' → Type):

```
Class URForall_Type_class A A' {dom : UR A A'} (P : A → Type) (Q : A' → Type) :=
 { transport_ :> Transportable P; ur_type :> ∀ x y (H:x ≈ y), P x ⋈ Q y }.

Definition URForall_Type A A' {HA : UR A A'} : UR (A → Type) (A' → Type) :=
  {| ur := fun P Q ⇒ URForall_Type_class A A' P Q |}.
```

This definition says that two predicates are in relation whenever they are in relation pointwise, and when P is transportable.

Using Transportable, we can instrument the definition of univalent relation on dependent products to improve effectiveness. More precisely, in the definition of the inverse function that defines the equivalence (∀ x : A, B x) ≃ (∀ x : A', B' x) we use the fact that B is transportable to change

the dependency in B instead of pattern matching on the equality between the dependencies. This is possible because from $e_B : B \approx B'$, we know that B is transportable (thanks to the specialized definition URForall_Type).

8 UNIVALENT PARAMETRICITY IN ACTION: EXPLAINED

We now come back to the examples of §4, explaining the definitions and adjustments of the typeclass resolution mechanism necessary to achieve seamless transport à la carte.

Adding a univalent relation in the global context. To declare that unary and binary natural numbers are univalently related, one first needs to provide a proof IsEquiv_of_ℕ that the transport function Bin.of_ℕ from ℕ to Bin is actually an equivalence, and declare it as a typeclass instance.

```
Instance Equiv_ℕBin : ℕ ≃ Bin := BuildEquiv ℕ Bin Bin.of_ℕ IsEquiv_of_ℕ.
```

Then, using for example the canonical relation fun (n:Bin) (m:ℕ) ⇒ n = ↑■ m to define the univalent relation UR_ℕ_Bin between ℕ and Bin, the only remaining piece is the proof of the following coherence condition, which can easily be done using the section of the equivalence.

```
Definition coherence_ℕBin : ∀ a a' : ℕ, (a = a') ≃ (a = Bin.to_ℕ (Bin.of_ℕ a')).
```

Then, one can define an instance of ℕ ⋈ Bin.

```
Instance univrel_ℕBin : ℕ ⋈ Bin :=
{| equiv := Equiv_ℕBin;
  Ur := UR_ℕ_Bin;
  Ur_Coh := {| ur_coh := coherence_ℕBin |}; |}
```

This way, typeclass resolution is able to automatically derive further instances of the relation based on this basic univalent relation, emulating the White Box FP of Corollary 5.3. For instance, Coq can automatically infer the relation between ℕ → ℕ → ℕ and Bin → Bin → Bin because it is equal to ↑□ (ℕ → ℕ → ℕ):

```
Goal ℕ → ℕ → ℕ ⋈ Bin → Bin → Bin.
 typeclasses eauto.
Qed.
```

Transport à la carte. However, the fact that ℕ and Bin are in univalent relation alone only provides black-box transport on functions manipulating integers. To get more efficient transport, one can do an additional proof effort in order to also get some white-box transport, exploiting the relation between some particular functions. For instance, one can prove that the addition functions on unary and binary numbers are univalently related:

```
Definition univrel_add : plus ≈ plus_Bin.
```

This amounts to show that for every n m : ℕ, we have plus n m = ↑■ (plus_Bin (↑■ n) (↑■ m)). The proof can be done by induction on n. The O case requires showing that plus_Bin O_Bin m = m for every m : Bin, which is true by computation. The S case requires showing that plus_Bin (S_Bin n) m =

S_{Bin} ($plus_{Bin}$ n m) for every n m: Bin. This property is more complex to prove because it must be done by induction on n and the definition of S_{Bin} does not comply very well with the binary structure.

Next, to add this relation to the global context, we need to instrument typeclass resolution by defining the following hint, which will be used when looking for a function in relation with the plus function:

```
Hint Extern 0 (plus _ _ ≈ _) ⇒ eapply univrel_add : typeclass_instances.
```

With this hint, the system is able to automatically infer that the types ∀ n m : ℕ, n + m = m + n and ∀ n m : Bin, n + m = m + n are in univalent relation using white-box transport. For instance:

```
Goal (∀ n m : ℕ, n + m = m + n) ⋈ (∀ n m : Bin, n + m = m + n).
 typeclasses eauto.
Qed.
```

It is therefore possible to automatically transport the *proof term* of the commutativity of plus to a proof term of the commutativity of $plus_{Bin}$ using the black-box transport provided by this univalent relation.

```
Definition plus_Bin_comm : ∀ n m : Bin, n + m = m + n := ↑■ plus_comm.
```

Transporting goals. As explained in §4, univalent parametricity can also be used to prove properties by computation using an alternative representation that is more adequate computationally. For instance with the polynomial poly, the proof that poly 50 is bigger than 1000 can be done by moving to an equivalent property on binary natural numbers first, and then solving the goal by computation and basic inversion.

```
Goal poly 50 ≥ 1000.
 replace_goal; now compute.
Defined.
```

The tactic replace_goal proceeds by first asserting that there exists a property opt that is in univalent relation with the given goal (here poly 50 ≥ 1000), and inferring this equivalent property through typeclass resolution using white-box transport. Then the equivalence induced by the univalent relation is used to replace the original goal with the inferred property opt; this is black-box transport. The definition of the replace_goal tactic is simple in Ltac:

```
Ltac replace_goal :=
 match goal with |   ⊢   ?P ⇒ let X := fresh "X" in
  refine (let X := _ : { opt : Prop & P ≈ opt} in _);
   [ eexists; typeclasses eauto | apply (e_inv (equiv X.2))]
 end.
```

It first introduces the definition of a property opt that is in univalent relation with P. This property opt is obtained automatically by triggering the typeclass resolution on P ≈ ?, finding a

canonical instance. If it succeeds, this step gives at the same time the definition of `opt` and the proof that it is in univalent relation with `P`. In particular, this proof contains an equivalence between `P` and `opt`, which is used to replace the goal `P` by `opt`, using the inverse function of the equivalence.

Fixpoints. The proof technique above also scales to fixpoints, even though fixpoints must be dealt with in a non-generic way. Concretely, one needs to provide a univalent parametricity instance for each case of pattern matching performed inside the fixpoint. In the sequence example of §4.3, the required instance must be defined for fixpoint matching on 0, 1, and 2:

```
Definition fix_ℕ3 :
 (fun P X0 X1 X2 XS ⇒ fix f (n : ℕ) {struct n} : P := match n with
                                                      | 0 ⇒ X0
                                                      | 1 ⇒ X1
                                                      | 2 ⇒ X2
                                                      | S n ⇒ XS n (f n) end) ≈
 (fun P X0 X1 X2 XS ⇒ fix f (n : ℕ) {struct n} : P := match n with
                                                      | 0 ⇒ X0
                                                      | 1 ⇒ X1
                                                      | 2 ⇒ X2
                                                      | S n ⇒ XS n (f n) end).
```

The proof of this instance is systematic, and can be done automatically using `induction` and `typeclasses eauto`. However, because pattern matchings are not first-class objects in Coq, it is not possible to define a single generic univalent parametric instance for every fixpoint. Note that this (practical, rather than theoretical) issue does not manifest when using eliminators, because there is only one eliminator per inductive type.

Limitations of the current Coq *implementation.* The use of typeclasses and typeclass resolution to deal with the global context of univalently related constants is both a blessing and a curse. It is nice because it allows us to instrument univalent parametricity in Coq without modifying the source code, offering great flexibility and accessibility. But this approach does not scale very well to large developments because typeclass resolution is internally based on proof search, which quickly becomes intractable. In practice, in our current implementation, we observe that successful typeclasss resolution is fairly fast, but when the proof search fails because of some missing `Hint`s, resolution can take a very long time or may even diverge.

This issue is a known limitation of using typeclasses to drive automatic program transformations, and can also be experienced in other frameworks like CoqEAL [Cohen et al. 2013]. It could be addressed via a direct implementation of univalent parametricity, for instance using MetaCoq [Sozeau et al. 2020a,b]. With a MetaCoq plugin, it is possible to have complete access to the reification of a term of Coq in Coq. This would provide complete programmatic control over the univalent parametricity translation, thereby avoiding issues that follow from relying on proof search. Another possibility is to implement the translation directly as a Coq plugin in OCaml, as has recently been done for the white-box approach by Ringer et al. [2019]. However, the direct definition of a plugin is very sensitive to changes in the implementation of the Coq proof assistant itself, so we believe that the MetaCoq approach would be better suited, as it provides an abstraction barrier between the theory of Coq and its actual implementation.

9 CASE STUDY: NATIVE INTEGERS

To further illustrate the applicability of univalent parametricity, we consider a case study based on a recent improvement to Coq: native 63-bits integers, available starting with Coq 8.10.[22] This extension raises the question of how to interface a native datatype within Coq, supporting reasoning about (and with) such native values.

Native integers provide a basic datatype `int` together with native functions such as the left lshift operator `a << b`, which shifts each bit in `a` to the left by the number of positions indicated by `b`. These are defined as follows:

```
Register int : Set as int63_type.
Primitive lsl := #int63_lsl.
Infix "<<" := lsl (at level 30, no associativity) : int63_scope.
```

Because the operations are native, there is no direct way to reason about them in Coq. This is why the standard library of Coq relates `int` to binary numbers $\mathbb{Z}$ (these are similar to Bin, but include negative numbers), and states *axioms* to specify the behavior of native functions.

```
Definition wB := 2 ^ 63.

Definition to_Z : int → Z := ... (* explicit definition using operations on int *)
Definition of_Z : Z → int := ... (* explicit definition using operations on int *)

Axiom of_to_Z : ∀ (x:int), of_Z (to_Z x) = x.
Axiom lsl_spec : ∀ x p, to_Z (x << p) = to_Z x * 2 ^ (to_Z p) mod wB.
```

The statements of `of_to_Z` and `lsl_spec` are very natural, but the first question it raises is about completeness: How can we be sure that these two axioms are enough to prove any property on `lsl`? For instance, do we also need to postulate that `to_Z` forms a retraction?

Actually, it is possible to derive the other part of the correspondence between `int` and $\mathbb{Z}$ (note that this is not an axiom, it is proven by induction on `z` in $\mathbb{Z}$):

```
Lemma of_Z_spec : ∀ (z:Z), to_Z (of_Z z) = n mod wB.
```

Considering `of_to_Z` and `of_Z_spec`, it would seem that `int` and $\mathbb{Z}$ are indeed univalently related and that functions on `int` can likewise be univalently related to functions on $\mathbb{Z}$. Actually, the careful reader might have noticed that `of_Z_spec` does not exactly correspond to the statement of a retraction on `to_Z`. This is because `int` is actually in relation with $\mathbb{Z}/2^{63}\mathbb{Z}$. Therefore, we can define $\mathbb{Z}/2^{63}\mathbb{Z}$ as the type $\mathbb{Z}$wB of integers between 0 and 2^{63}, and adjust `to_Z` and `of_Z` accordingly.

```
Definition ZwB := { n : Z & 0 ≤ n < wB }.

Lemma to_Z_bounded : ∀ x, 0 ≤ to_Z x < wB.
Definition to_ZwB : int63 → ZwB := fun x ⇒ (to_Z x; to_Z_bounded x).
Definition of_ZwB (z:ZwB) : int63 := of_Z z.1.
```

[22] See https://github.com/coq/coq/blob/v8.10/theories/Numbers/Cyclic/Int63/Int63.v

The axioms of_to$_{\mathbb{Z}}$ and lsl_spec are exactly what is required to show that int and $\mathbb{Z}$wB are univalently related and that the native function lsl is univalent related to the corresponding function on $\mathbb{Z}$wB.

```
Definition IsEquiv_to_ℤ_ : IsEquiv to_ℤwB.
Instance univrel_int_ℤwB : int ⋈ ℤwB.

Definition ℤwB_lsl : ℤwB → ℤwB → ℤwB :=
 fun n m ⇒ ((n.1 * ℤ.pow 2 m.1) mod wB ; (* easy proof term omitted *)).

Notation "n << m" := (ℤwB_lsl n m) : ℤwB_scope.

Definition univrel_lsl : lsl ≈ ℤwB_lsl.
```

We have illustrated the correspondence between int and $\mathbb{Z}$wB using the lsl function, but the very same can be done for all functions of the native integers interface.

Once this is done, it is possible to use univalent parametricity to go beyond what is currently provided in the Coq standard library, such as proving concrete properties on $\mathbb{Z}$wB using an automatic transport to int. Consider the following polynomial, and two similar proofs by computation: the first directly on $\mathbb{Z}$wB, and the second on int after transport.

```
Definition poly_ℤ : ℤwB → ℤwB := fun n ⇒ 45 + ℤwB_pow n 100 - ℤwB_pow n 99 * 16550.

Goal poly_ℤ 16550 = 45.
 Time reflexivity.
Defined.

Goal poly_ℤ 16550 = 45.
 replace_goal. Time reflexivity.
Defined.
```

While both executions of reflexivity terminate, the execution time when the goal is not shifted to int is two orders of magnitude slower than when it is (0.3s vs. 0.002s). The difference for this precise (artificial) example may not seem that significant in absolute terms, but we can expect it to be interesting in large-scale developments, which could justify the use of native integers.

The second—maybe more important—advantage of organizing all the axioms on the specification of the functions on native integers using univalent parametricity is that it guarantees *completeness* of the axiomatization. Indeed, by the White Box FP (Corollary 5.3), we are certain that any theorem on int and its native functions is univalently related to a theorem on $\mathbb{Z}$wB. And by the Black Box FP (Proposition 5.4), such univalent relation allows us to easily transport the proof of this theorem on $\mathbb{Z}$wB to a proof of the corresponding theorem on int. For instance, the distributivity of << over addition can be automatically transported from $\mathbb{Z}$wB to int:

```
Definition ℤwB_lsl_add_distr x y n : (x + y) << n = (x << n) + (y << n).
(* proof using properties of mod and automation on ℤ *)
```

```
Definition lsl_add_distr : ∀ x y n, (x + y) << n = (x << n) + (y << n) :=
 ↑▪ ℤwB_lsl_add_distr.
```

In contrast, the proof of `lsl_add_distr` in the Coq standard library is done manually, and in fact does not even use the auxiliary lemma `ℤwB_lsl_add_distr`. Instead, the proof is dealing with both the conversion to $\mathbb{Z}$ and the proof of the property on $\mathbb{Z}$ at the same time. We believe that systematically proving properties first on `ℤwB`, and then automatically transporting them to `int` simplifies development, maintenance, and understanding.

10 RELATED WORK

Type theories. Homotopy Type Theory [Univalent Foundations Program 2013] treats equality of types as equivalence. For regular datatypes (also known as homotopy sets or hSets), equivalence boils down to isomorphism, hence the existence of transports between the types. However, as univalence is considered as an axiom, any meaningful use of the equality type to transport terms along equivalences results in the use of a non-computational construction. In contrast, in this work we carefully delimit the effective equivalence-preserving type constructors in our setting, pushing axioms as far as possible, and supporting specialized proofs to avoid them in certain scenarios.

Cubical Type Theory [Cohen et al. 2015] provides computational content to the univalence axiom, and hence functional and propositional extensionality as well. In this case, the invariance of constructions by type equivalence is built in the system and the equality type reflects it. Note that the recent work of Altenkirch and Kaposi [2015] on a cubical type theory without an interval proposes to use a relation quite close to the one defined in univalent parametricity to encode equality in the theory. They are handling a different problem (albeit in a similar way), because they are trying to build a theory that supports univalence. In our framework, we relate the relation to equality and type equivalence, which allows us to stay within CIC, without relying on another more complex type theory, but the price we pay is to assume univalence as an axiom. Note that while we focus on CIC extended with the univalence axiom, univalent parametricity could be likewise developed in a cubical theory, such as Cubical Agda [Vezzosi et al. 2019]. The only change is that the definition of univalent relations for types and functions can make use of top-level equality directly instead of using explicit extensional definitions such as type equivalence and pointwise equality. In particular, terms such as Equiv_Π or univ_Π (§5) can be largely simplified. But the interest of the general relational univalent parametric setting remains unchanged because it provides heterogeneous automatic transport, which is not readily available in cubical type theories.

Observational Type Theory (OTT) [Altenkirch et al. 2007] uses a different notion of equality, coined John Major equality. It is a heterogeneous relation, where terms in potentially different types can be compared, usually with the assumption that the two types will eventually be *structurally* equal, not merely equivalent. This stronger notion of equality of types is baked in the type system, where type equality is defined by recursion on the structure of types, and value equality follows from it. It implies the K axiom, which is in general inconsistent with univalence, although certainly provable for all the non-polymorphic types definable in OTT. A system similar to ours could be defined on top of OTT to allow transporting by equivalences.

Parametric Type Theory and the line of work integrating parametricity theory to dependent type theory, either internally [Bernardy et al. 2015] or externally, is linked to the current work in the sense that our univalent parametricity translation is a refinement of the usual parametricity translation. In its simple form, parametricity in type theory does not admit an identity extension lemma, which ensures that if we pass the identity type as relations for the arguments of type constructors, then the resulting relation for the type constructor is equivalent to the identity. This

issue has been addressed, first on small types [Atkey et al. 2014] by considering the reflexive graph model and then on an extension of type theory with a parametric function type [Nuyts et al. 2017]. In our work, we get a variant of the identity extension lemma very easily because all the relations we consider need to be related to equality through the coherence condition. However, we do not exactly get identity extension: for instance on the type of booleans $\mathbb{B}$, it is also possible to provide the relation which says that `true` is related to `false` (and dually), together with the equivalence which flips booleans. This defines a univalent relation on $\mathbb{B}$ because it satisfies the coherence condition, but the relation does not coincide with equality on booleans—it only does up to flipping booleans.

Recently, Cavallo and Harper [2020] proposed a theory mixing both cubical type theory and parametric type theory. However, they are not considering univalent parametricity, but rather a theory where proofs of parametricity can make use of univalent principles, such as functional extensionality or the univalence axiom.

For Extensional Type Theory, Krishnaswami and Dreyer [2013] develop an alternative view on parametricity, more in the style of Reynolds, by giving a parametric model of the theory using quasi-PERs and a realizability interpretation of the theory. From this model construction and proof of the fundamental lemma they can justify adding axioms to the theory that witness strong parametricity results, even on open terms. However they lose the computability and effectiveness enjoyed by both Bernardy's construction and ours, which are developed in intensional type theories.

The parametricity translation of Anand and Morrisett [2017] extends the logical relation on propositions to force that related propositions are *logically* equivalent. It can be seen as a degenerate case of our approach that forces related types to be equivalent, considering that equivalence boils down to logical equivalence on propositions (see §5.2.1 for a more detailed explanation). However the translations differ in other aspects. While our translation requires the univalence axiom, theirs assumes proof irrelevance and the K axiom, and does not treat the type hierarchy. Our solution to the fixpoint arising from interpreting $\mathtt{Type}_i : \mathtt{Type}_{i+1}$ is original, along with the use of conditions to ensure coherence with equality. They study the translation of inductively-defined types and propositions in detail, giving specific translations in these two cases to accommodate the elimination restrictions on propositions, and are more fine-grained in the assumptions necessary on relations in parametricity theorems. In both cases, the constructions were analyzed to ensure that axioms were only used in the non-computational parts of the translation, hence they are effective.

Relational parametricity in type theory can be understood categorically in terms of inverse diagrams. Indeed, given a category with attributes (CwA) C, the category of inverse diagrams $C^{\mathbf{Span}}$ (where **Span** is the "walking Span" category : $0 \leftarrow 01 \rightarrow 1$) is also a CwA where a type is now a triple of two types and a relation between them. This interpretation does not extend directly to univalent parametricity but the notion of homotopical inverse diagrams recently developed by Kapulkin and Lumsdaine [2018] seems to provide a categorical interpretation of univalent parametricity—namely, as inverse diagrams over the category $\cdot \rightarrow \cdot$ with two objects and one arrow between them which is "marked" as an equivalence. Kapulkin and Lumsdaine [2018] show that under the assumption that the CwA C satisfies functional extensionality, the category $C^{\cdot\rightarrow\cdot}$ is also a CwA. However, they do not investigate the interpretation of universes and univalent universes as done for instance by Shulman [2015]. Our work suggests that the interpretation of universes for homotopical inverse diagrams can only be done for univalent universes, in the same way as dependent products can only be lifted in presence of functional extensionality.

Data refinement. Another part of the literature deals with the general data refinement problem, e.g. the ability to use different related data structures for different purposes: typically simplicity of

proofs versus efficient computation. The frameworks provide means to systematically transport results from one type to the other.

Magaud and Bertot [2000] and Magaud [2003] first explored the idea of transporting proof terms from one data representation to another in Coq, assuming that the user provides a translation of the definitions from one datatype to the other. It is limited to isomorphism and implemented externally as a plugin. The technique is rather invasive in the sense that it supports the transport of proof terms that use the computational content of the first type (*e.g.,* the reduction rules for `plus` on natural numbers) by making type conversions explicit, turning them into propositional rewrite rules. This approach breaks down in presence of type dependencies.

In CoqEAL [Cohen et al. 2013] refinement is allowed from proof-oriented data types to efficiency-oriented ones, relying on generic programming for the computational part and automating the transport of theorems and proofs. CoqEAL does not only deal with isomorphisms, but also quotients, and even partial quotients, which we cannot handle. The approach exploits parametricity for generating proofs, but it does not support general dependent types, only parametric polymorphism. Moreover, the advocated style prevents doing local transport and rather requires working with interfaces from the outset—which we coined the *anticipation problem*—and applying parametricity in a second step. We can avoid anticipating common interfaces thanks to our limitation to transport by equivalences.

In a categorical setting, Robinson [1994] uses parametricity for System F to show that when a function is defined on a type A using only the fact that it is isomorphic to a given type B, then the function can be transported to another type A' as long as it is also isomorphic to B. This approach to parametricity is similar to that of CoqEAL, which uses the fact that when working with an abstract interface—here an abstract copy of a type—the term does not depend on the implementation of the interface. This is different from the univalent parametricity approach we develop here, which works on two concrete types and functions defined on them, which can *a posteriori* be shown equivalent.

Haftmann et al. [2013] explain how the Isabelle/HOL code generator uses data refinements to generate executable versions of abstract programs. The refinement relation used is similar to the partial quotients of CoqEAL. The Autoref tool for Isabelle [Lammich 2013] also uses parametricity for refinement-based development. It is an external tool that synthesizes executable instances of generic algorithms and refinement proofs.

Huffman and Kunčar [2013] address the problem of transferring propositions between different types, typically a representation type (*e.g.,* integers) to an abstract type (*e.g.,* natural numbers) in the context of Isabelle/HOL. Again this allows to relate a type and its quotient, like in CoqEAL, and is based on parametricity. Recently, Zimmermann and Herbelin [2015] present an algorithm and plugin to transport theorems along isomorphisms in Coq similar to that of Huffman and Kunčar [2013]. In addition to requiring the user to provide a surjective function f to relate two data types, their technique demands that the user explicitly provide transfer lemmas of the form $\forall x_1 \dots x_n,\ R(x_1 \dots x_n) \implies R'(f(x_1) \dots f(x_n))$, for each relation R that the user expects to transfer to a relation R'. The approach is not yet able to handle parameterized types, let alone dependent types and type-level computation.

Recently, Ringer et al. [2019] developed a tool in Coq to automatically build equivalences between inductive types using the theory of ornaments [Dagand and McBride 2014]. These equivalences are then instrumented to transport functions and proofs using a framework that is largely inspired by our previous conference article on univalent parametricity [Tabareau et al. 2018]. However, the approach is implemented as a Coq plugin in OCaml. While this can be convenient to achieve full automation without relying on the somewhat brittle typeclass mechanism, it also presents major risks by being tied to a specific implementation of Coq. A direct implementation based on MetaCoq [Sozeau et al. 2020a,b] would seem preferable. Independently of the implementation

strategy, the tool developed by Ringer et al. [2019] is essentially the white-box transport described in this article. Because they never perform black-box (univalent) transport, however, they also encountered the computation problem of parametricity described in § 2.3. We hope that the many clarifications provided in this extended and revised article will prove helpful for addressing these issues.

11 CONCLUSION

We have presented *univalent parametricity*, a fruitful marriage of parametricity and univalence that fully realizes programming and proving modulo equivalences in type-theoretic proof assistants. Univalent parametricity supports two complementary reasoning principles and forms of transport, resulting in *transport à la carte*: from a type equivalence, univalent transport operates in a black-box manner; additional proofs of equivalences between functions over related types allow heterogeneous parametricity to transport terms in a white-box manner up to these equivalences. We have shown how this makes it possible to conveniently switch between an easy-to-reason-about representation and a computationally-efficient representation. Our approach is realizable even in type theories where univalence is taken as an axiom, such as in Coq, and we have explored how to maximize the effectiveness of transport in this setting. Several examples and use cases in Coq attest to the practical impact of this work to provide easier-to-use proof assistants by supporting seamless programming and proving modulo equivalences.

ACKNOWLEDGMENTS

We thank the reviewers of the original ICFP 2018 publication, as well as the ICFP 2018 participants for asking several questions that forced us to better understand the limits of our initial proposal. This eventually led us to the enhanced presentation of this article. We are thankful as well to Assia Mahboubi for suggesting concrete applications for validating the use of our approach, which likewise pushed us to extend and enhance this work on univalent parametricity. Finally, we are extremely grateful to the anonymous JACM reviewers for their acute and eminently helpful comments and suggestions, which have resulted in considerable clarifications and more precise expositions of the concepts.

REFERENCES

Thorsten Altenkirch and Ambrus Kaposi. 2015. Towards a Cubical Type Theory without an Interval. In *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, Tarmo Uustalu (Ed.), Vol. 69. LIPICS.

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational equality, now!. In *Proceedings of the Workshop on Programming Languages meets Program Verification (PLPV 2007)*. 57–68.

Abhishek Anand and Greg Morrisett. 2017. Revisiting Parametricity: Inductives and Uniformity of Propositions. *CoRR* abs/1705.01163 (2017).

Carlo Angiuli, Kuen-Bang Hou, and Robert Harper. 2018. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. 6:1–6:17.

Robert Atkey, Neil Ghani, and Patricia Johann. 2014. A Relationally Parametric Model of Dependent Type Theory. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 503–515.

Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. 2017. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. ACM, New York, NY, USA, 164–172.

Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. 2015. A Presheaf Model of Parametric Type Theory. *Electronic Notes in Theoretical Computer Science* 319 (2015), 67–82.

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming* 22, 2 (March 2012), 107–152.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.). 2013. *Proceedings of the 4th International Conference on Interactive Theorem Proving (ITP 2013)*. Lecture Notes in Computer Science, Vol. 7998. Springer-Verlag.

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Certified Programs and Proofs (CPP 2017)*. Paris, France, 182 – 194.

Evan Cavallo and Robert Harper. 2020. Internal Parametricity for Cubical Type Theory. , 13:1–13:17 pages.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2015. Cubical Type Theory: a constructive interpretation of the univalence axiom. (May 2015), 5:1–5:34 pages.

Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *Proceedings of the International Conference on Certified Programming and Proofs (CPP 2013) (Lecture Notes in Computer Science)*, G. Gonthier and M. Norrish (Eds.), Vol. 8307. Springer-Verlag, 147–162.

The Coq Development Team. 2019. *The* Coq *proof assistant reference manual*. http://coq.inria.fr Version 8.10.

Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2-3 (Feb. 1988), 95–120.

Pierre-Évariste Dagand and Conor McBride. 2014. Transporting functions across ornaments. *Journal of Functional Programming* 24, 2-3 (2014), 316–383.

Nicola Gambino and Martin Hyland. 2004. Wellfounded trees and dependent polynomial functors. In *Proceedings of Types for Proofs and Programs (TYPES 2003) (Lecture Notes in Computer Science)*, Vol. 3085. Springer-Verlag, 210–225.

Healfdene Goguen, Conor McBride, and James McKinna. 2006. *Eliminating Dependent Pattern Matching*. Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540.

Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. 2013. Data Refinement in Isabelle/HOL, See [Blazy et al. 2013], 100–115.

Michael Hedberg. 1998. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming* 8, 4 (July 1998), 413–436.

Brian Huffman and Ondřej Kunčar. 2013. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Proceedings of the 3rd International Conference on Certified Programs and Proofs (CPP 2013)*. Springer-Verlag, Melbourne, Australia, 131–146.

Chris Kapulkin and Peter LeFanu Lumsdaine. 2018. Homotopical inverse diagrams in categories with attributes. *arXiv preprint arXiv:1808.01816* (2018).

Neelakantan R. Krishnaswami and Derek Dreyer. 2013. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In *Proceedings of the Conference for Computer Science Logic (CSL 2013)*. 432–451.

Peter Lammich. 2013. Automatic Data Refinement, See [Blazy et al. 2013], 84–99.

Nicolas Magaud. 2003. Changing Data Representation within the Coq system. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003) (Lecture Notes in Computer Science)*, D. Basin and B. Wolff (Eds.), Vol. 2758. Springer-Verlag.

Nicolas Magaud and Yves Bertot. 2000. Changing Data Structures in Type Theory: A Study of Natural Numbers. In *International Workshop on Types for Proofs and Programs (TYPES 2000) (Lecture Notes in Computer Science)*, P. Callaghan, Z. Luo, J. McKinna, and R. Pollack (Eds.), Vol. 2277. Springer-Verlag, 181–196.

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73 – 118.

Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, 1–2.

Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric Quantifiers for Dependent Type Theory. *Proc. ACM Program. Lang.* 1, ICFP, Article Article 32 (Aug. 2017), 29 pages.

Christine Paulin-Mohring. 2015. Introduction to the Calculus of Inductive Constructions. In *All about Proofs, Proofs for All*, Bruno Woltzenlogel Paleo and David Delahaye (Eds.). Studies in Logic (Mathematical logic and foundations), Vol. 55.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN Conference on Functional Programming (ICFP 2006)*. ACM Press, Portland, Oregon, USA, 50–61.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for Proof Reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:19.

Edmund Robinson. 1994. Parametricity as isomorphism. *Theoretical Computer Science* 136, 1 (1994), 163 – 181.

Michael Shulman. 2015. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science* 25, 5 (2015), 1203–1277.

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020a. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 8:1–8:28.

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020b. The MetaCoq Project. *Journal of Automated Reasoning* (Feb. 2020).

Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proceedings of the ACM on Programming Languages* 2, ICFP (Sept. 2018), 92:1–92:29.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (Aug. 2019).

Vladimir Voevodsky. 2010. The Equivalence Axiom and Univalent Models of Type Theory. arXiv:1402.5556.

Philip Wadler. 1989. Theorems for Free!. In *Functional Programming Languages and Computer Architecture*. ACM Press, 347–359.

Theo Zimmermann and Hugo Herbelin. 2015. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. arXiv:1505.05028v4.